CrossTalk

March / April 2011 The Journal of Defense Software Engineering Vol. 24 No. 2

THE RUGGED SOFTWARE MANIFESTO

I am rugged—and more importantly, my code is rugged.

I recognize that software has become a foundation of our modern world.

I recognize the awesome responsibility that comes with this foundational role.

I recognize that my code will be used in ways I cannot anticipate, in ways it was not designed, and for longer than it was ever intended.

I recognize that my code will be attacked by talented and persistent adversaries who threaten our physical, economic, and national security.

I recognize these things—and I choose to be rugged.

I am rugged because I refuse to be a source of vulnerability or weakness.

I am rugged because I assure my code will support its mission.

I am rugged because my code can face these challenges and persist in spite of them.

I am rugged, not because it is easy, but because it is necessary...and I am up for the challenge.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAR 2011		**00-03-2011 to 00-04-2011**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk. The Journal of Defense Software Engineering. Volume 24, Number 2, March/April 2011	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS MXDEA,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**36**	

Cover Design by Kent Bingham

Departments

RUGGED SOFTWARE

CrossTalk

OSD (AT&L) Stephen P. Welby
NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Advisor Kasey Thompson
Article Coordinator Lynne Wade
Managing Director Brent Baxter
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-775-5555
E-mail stsc.customerservice@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the Under Secretary of Defense for Acquisition, Technology and Logistics (USD(AT&L)); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Defense (DHS). USD(AT&L) co-sponsor: Director of Systems Engineering. USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division in the National Protection and Program Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of **CrossTalk** providing both editorial oversight and technical review of the journal. **CrossTalk's** mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the **CrossTalk** editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. **CrossTalk** does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with **CrossTalk.**

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of **CrossTalk** are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the CrossTalk webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <http://www.luminpublishing.com>.

CrossTalk would like to thank the DHS for sponsoring this issue.

Vulnerabiquity and the Value of Rugged Infrastructure

As I write this **CrossTalk** introduction, we're coming off of a nightmarish patch week with over 120 patches streaming from industry. Despite SQL injection celebrating its 10th birthday, the 2010 Verizon Business Data Breach Investigations Report clocked its use in 89% of last year's breached records. Our reality is that software vulnerabilities are ubiquitous. Our reality is a state of Vulnerabiqiuty. Our reality leaves much room for improvement–and yet I remain optimistic.

In the physical world, we've come to depend on concrete, steel, and iron to support our bridges, our tunnels, our skyscrapers … Architects, engineers, and the like carry an awesome responsibility to provide safe and reliable foundations for society and our lives. Chances are you aren't living in perpetual fear that your office building will collapse upon you. For the most part, our infrastructure is dependable.

This is not the case with our digital infrastructure and software. Software has increasingly become modern infrastructure. Ones and zeros permeate most aspects of our daily existence. Yet software is not nearly as trustworthy or reliable as our physical infrastructure. For those in security, we've become painfully aware of the seemingly infinite vulnerabilities in digital infrastructure. The Information Superhighway is a battlefield. Practitioners spend millions every year deploying critical patches to fragile software. Criminals leverage software weaknesses to perpetrate breaches of identity, credit card data, and intellectual property. Moreover, nation states exploit weak software threatening our physical, economic, and national security. Since we depend upon software as foundational infrastructure, it must be up for the task.

We need better. We deserve better. Thanks to a focus on value, we may be on the cusp of bringing about substantive change. People seek what they value. We reward and measure what we value. We demand value. Where there is demand, supply will follow. Rugged is an affirmative value–not a cost or inhibitor. Rather than focus on supply, Rugged articulates and fosters demand. Rugged is comprehended and coveted by business people. Rugged evokes our desire for available, survivable, supportable, defensible infrastructure for what matters most to us. While others look for the worst in developers, Rugged seeks to encourage the best in them. To date, software security has focused on technology. Whoever coined "People, Process, and Technology" was brilliant to put people first, and technology last. We've skipped people and values–and as a consequence software security has remained relegated to the .0001% of the greater industry. Success will require ubiquitous demand for the value of Rugged infrastructure–the supply will follow.

To my delight, we've realized our Rugged Manifesto <http://www.ruggedsoftware.org> was a bit off the mark. While it successfully speaks to the hearts and values of many developers, the most dramatic successes have instead been on the demand side. People want and deserve better and more reliable digital infrastructure than today's status quo. They have taken Rugged to greater heights. We've seen tangible examples of where "security" has failed but "Rugged" has borne fruit.

Buyers are seeking more Rugged infrastructure. Employers are seeking more Rugged developers. Inspired neophytes are seeking Rugged training. Rugged is not a spectator sport. It needs you. This is the beginning. Please take to heart the articles in this issue as they address the problem at hand and let's collectively drive an end to the era of Vulnerabiquity. How will you help us to drive toward the Rugged Age?

Joshua Corman
Research Director for Enterprise Security at The 451 Group
Co-Founder of Rugged Software

Crumple Zones

Absorbing Attack Effects Before They Become a Problem

Michael Atighetchi, Raytheon BBN Technologies
Partha Pal, Raytheon BBN Technologies
Aaron Adler, Raytheon BBN Technologies
Andrew Gronosky, Raytheon BBN Technologies
Fusun Yaman, Raytheon BBN Technologies
Jonathan Webb, Raytheon BBN Technologies
Joe Loyall, Raytheon BBN Technologies
Asher Sinclair, U.S. Air Force Research Laboratory
Charles Payne, Adventium Labs

Abstract. A specific and currently relevant issue motivating the notion of ruggedized software is the confluence of the threat of cyber attacks and our increased dependence on software systems in enterprise as well as tactical situations. Software services that are essential for mission success must not only withstand normal wear and tear, stresses and accidental failures, they also must endure the stresses and failures caused by malicious activities and continue to remain usable. The Crumple Zone (CZ), a software shock absorber that absorbs attack effects before they cause significant system failures, is an architectural construct that we have developed and are maturing iteratively. We argue that the CZ is an important building block for constructing ruggedized software for supporting network-centric operations. In this paper we discuss the CZ in the context of Service-Oriented Architecture (SOA) and describe a configuration that has been realized and demonstrated.

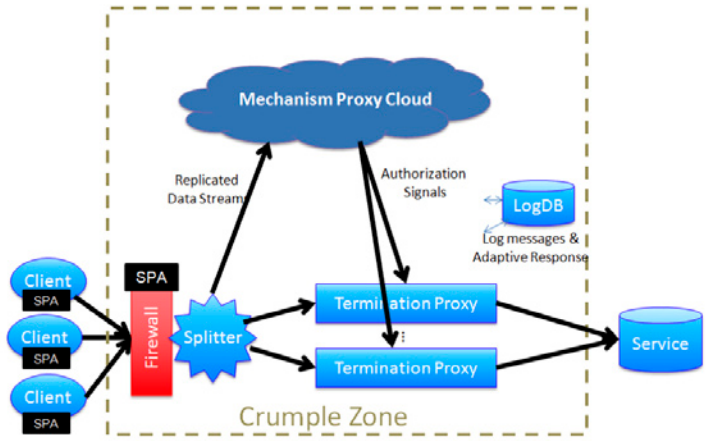
1. Introduction

A higher level of structural and operational endurance and ruggedness can be achieved in software systems by strategically introducing CZs in the system architecture. Analogous to the crumple zone in an automobile, a CZ stands before critical components and "absorbs" the effects of attacks by localizing or eliminating the damage they can cause and leaving the critical components intact and unaffected. The concept of software CZ is broadly applicable; in this paper we discuss CZs for SOA.

SOA is an architecture paradigm gaining popularity in military and civilian information systems, many of which play important roles in national security. Mission critical systems face a highly contested and hostile environment in real-world operations, and must endure and withstand malicious attacks. Potential threats against critical SOA-based systems range from automated network worms targeting SOA platform and supporting services to individual vandals to well-motivated and expert foreign intelligence apparatus that aim to subvert operations in the DoD enterprise and critical missions. The adversary objective may range from denying access to the system, to using the system without authorization, to tampering with or fabricating data in storage or in transit. But all indications, including our own assessment [1], point to serious lapses in the state of the art in SOA security. As a technology, SOA is still maturing and various aspects of SOA, including security features, are still being standardized. Furthermore, available SOA infrastructure and platforms do not always implement all of the available and specified standards. The complexity of SOA platforms combined with their rapid evolution can lead to implementers under-using or misusing available security features due to lack of expertise. Security of SOA systems is often limited to perimeter and network level [2] security.

Some of the very features that make SOA appealing, like loose coupling, dynamism, and composition-oriented system construction, make securing service-based systems more complicated. These features ease the development of systems, but also introduce additional vulnerabilities and points of entry than in self-contained, static, or stove-piped systems. In SOA, services are advertised and are looked up by potential users, many of which might not have the proper authorization to access or use the requested services. It is difficult to predict at design time exactly which actors will attempt to consume a given service and whether they will be authorized to do so. There are various system boundaries with a trust differential—one side is more trustworthy than the other side. Network and perimeter security only reinforce the "crunchy on the outside, chewy inside" view of software systems, and is utterly insufficient for developing rugged SOA systems.

Figure 1: Architectural Elements of the CZ



We argue that CZs can absorb attacks and minimize damage. CZs can be deployed at any trust boundary in the system. One key place we have experimented with and will describe in this paper is in the Demilitarized Zone (DMZ) between the services enclave and the public network from which clients access the services.

The rest of the paper is organized as follows. Section 2 provides an overview of the CZ architecture. Sections 3-7 describe various key features of the CZ and the components and mechanisms responsible for them. Section 8 describes Related Work, Section 9 provides performance metrics and a cost/benefit analysis. Section 10 concludes the paper.

2. CZ Architecture

The CZ is, in basic terms, a layer of intelligent service proxies that work together to present a high barrier to entry to the adversary, to increase the chance of detection of malicious activities, and to contain and recover from failures and undesired conditions caused by malicious attacks. These proxies collectively implement the service's consumer-facing application programming interface. Different proxies help contain malicious activity by applying security checks and controls, then approving data for release if it passes those checks. A key principle of the CZ's design is that only data that has been inspected and approved by one or more proxies is passed along to the service. Because the CZ inspects and processes untrusted data, it is expected to fail occasionally. Automatic monitoring and re-start of the proxies inside the CZ is another key design feature.

Effectiveness of the CZ depends on three requirements:

- The CZ must be non-bypassable. All consumer requests to the service must be mediated through the CZ.
- The CZ must cover both known and unknown attacks. It should be configurable so defenses can be tailored to the system's operational requirements and the potential threat environment.
- The CZ must preserve the integrity of data that flows through it to prevent man-in-the-middle scenarios run by corrupted CZ components.

To meet the first requirement, making the CZ non-bypassable, conventional network level protections such as firewalls and routers can be used. To make it difficult for adversaries to discover and access protected services, CZ presents a very small exploitable surface to untrusted service consumers. This is accomplished by placing the CZ behind a firewall that uses single packet authorization (SPA). On the CZ's side of the firewall, termination proxies (TPs) are used as the entry point for all incoming client connections.

The second requirement, varied and configurable defenses, is achieved through a set of proxies that implement specific checks and are organized in a mechanism proxy cloud (MPC). The MPC monitors observable behavior of requests. We have implemented proxies that check assertions on application data, e.g., by checking serialization fields, as well as canary proxies that consume application data and thereby absorb attacks, e.g., by crashing or getting corrupted.

The third requirement, preserving data integrity within the CZ, is achieved by service layer virtual private groups (sIVPG). The Splitter component replicates Secure Sockets Layer (SSL) streams between clients and TPs to the MPC without breaking cryptographic envelopes. Key management components that are also part of the sIVPG selectively share keys from the TPs to the MPC so that the new streams can be decrypted for inspection.

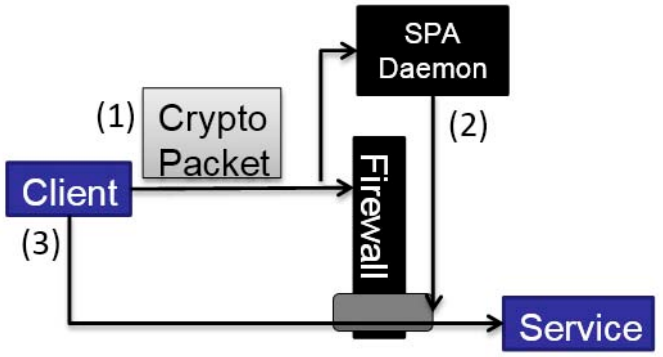
3. SPA

The first layer of defense an attacker coming from the outside needs to overcome is the CZ's firewall. In addition to standard restrictions on open ports and IP ranges, we use SPA [3] to implement a least-privilege policy that allows access to listening ports only to authenticated clients.

Figure 2 illustrates the general concept behind SPA using a client (on the left) trying to access the service (on the right) through the firewall (in the middle). The firewall starts out by blocking all traffic to the service. A legitimate client starts the interaction sequence (in step 1) by sending a cryptographic-based credential that is encoded within a single packet to the firewall. After verifying client identity and authorizing the client's connection request, the SPA server side component grants the client the right to establish a single Transmission Control Protocol (TCP) connection (for a limited amount of time) by adding specific firewall rules (step 2). Finally, the client establishes a normal TCP connection in step 3. A client without the proper credential is denied access.

SPA limits exposure of the protected enclave to port scans, remote OS fingerprinting, and low-level network stack exploits (such as TCP connection flooding). Port scan or OS fingerprinting attempts for reconnaissance will return no information unless the adversary has stolen or forged cryptographic credentials.

Figure 2: SPA



4. TP

TPs are advertised as service endpoints for the client, while the actual service is accessible only from the TP. The client believes it is connecting directly to the service, but the TP provides a barrier between the service and the client. The TP escrows client-server data until it is analyzed and determined to be safe to release.

One key design decision for constructing the TP was to keep its logic minimal and therefore make it less prone to exploits. For that reason, the TP does not itself analyze any client data because the analysis process might introduce corruption or crash faults. Instead, data analysis is performed in the MPC (see Section 5). If traffic passes all checks, the MPC sends authorization messages to the TP stating how many bytes of client data have been approved for release. The TP requires active approval of client data by the MPC within a certain amount of time. If the MPC detects anything wrong with the data or if the MPC fails to send a timely approval message, the connection to the client is closed by the TP and the escrowed data is discarded. Alternatively, when the MPC approves a certain number of bytes for release, the TP releases that amount of data from

escrow and sends it to the service. One key benefit of the split check-escrow model is that corrupted nodes in the MPC cannot directly affect the integrity of the application stream since MPC nodes only operate on a copy of the data and cannot alter the data that is released from the TP's escrow buffer. On the other hand, corrupted nodes in the MPC can incorrectly approve or disapprove release of escrowed data because the TP only receives instructions to release a certain number of bytes. This issue is dealt with by using voting on the release instruction, described in Section 5.

Crashes in the MPC will prevent approval messages from reaching the TP and will then result in the TP closing the connection to the client. All incoming client connections are routed through the TP–if the TP were to crash, many client connections would be terminated. Isolating the possible crashes in the MPC limits the number of clients affected by any crashes. Watchdogs help the system recover from crashes and are discussed in more detail in Section 7.

A single TP would be a single-point-of-failure in the CZ. This can be addressed by incorporating multiple TPs in the CZ, deployed in a manner analogous to load balancing. This provides isolation and replication to this critical part of the CZ. Additionally, in conjunction with the watchdog for the TP, the TPs can be moved and restarted to provide additional fault tolerance.

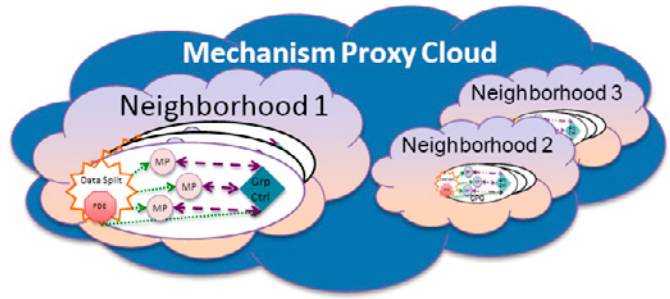
5. MPC

The MPC is a metaphor for a loosely coupled set of proxies that perform checks on application data. Figure 3 shows a detailed version of the MPC, which has a hierarchical structure. At the bottom of the hierarchy there are individual mechanism proxies (MPs) implementing check functionality, the next level up are the proxy groups (PGs), and finally the neighborhoods.

MPs inspect the content of the incoming traffic for attacks. For example, a rate proxy will raise a flag if the session has an unusually high message rate. Similarly a size proxy will reject a message with huge user data. Such proxies are useful for detecting known attacks, i.e., high message rate leading to denial of service, and big objects leading to heap overflow. To protect against novel attacks we utilize MPs that simulate (to a certain extent) the behavior of the protected service. If the simulated behavior is close enough to the actual behavior the effects of the novel attack can then be detected, absorbed, and managed by the proxy. The Canary proxy is an example based on this technique. Like the historical canary in a coalmine, a canary proxy will be affected by the attack in the same way the protected entity would. Canary is designed to parse the incoming stream the same way the server would thus protecting the deployed service against attacks that might be caused by arbitrarily malformed streams or arbitrary attack commands encoded in serialized data (for example, serialized instances of Java classes).

PGs represent a coordinated collection of MPs that together perform checks on application traffic. PGs are associated with SSL connections; each SSL connection between clients and TPs will be forwarded (through the sIVPG) to a dedicated PG. This assignment can be controlled at runtime based on available resources. The proxies within a group coordinate with a group controller (one controller per group), which regulates the control flow between the proxies in the group. Intuitively, the group controller enforces an order of execution on the proxies for improved protection. For example, to prevent unnecessary deaths of the canary proxy, we can chain a blacklist proxy, which screens for instances of known malicious classes, before the canary. The group controller is also responsible for communicating with the TP to notify it of the number of bytes cleared by all of the proxies in the group.

Figure 3: SPA



Neighborhoods represent fault isolation boundaries and are associated with processes in the current implementation model. For example, a corrupted MP running in Neighborhood 1 cannot directly access or spread to other MPs running in Neighborhood 2. A neighborhood can host multiple groups for load balancing purposes. Neighborhoods can be distributed within the MPC on different physical hosts and virtual machines.

In most cases, the crash of a canary like proxy also implies the crash of all components in the same neighborhood. This means that sessions of all clients sharing the same neighborhood will terminate. However, clients connecting through other neighborhoods will not be affected and future connections will go through the remaining neighborhoods.

To address the issue of a malicious MP incorrectly instructing the TP about escrow release mentioned earlier, one needs to assign redundant PGs to a single SSL connection and vote on the group's release decision. If the PGs are sufficiently independent, known fault tolerance schemes can be employed to detect and tolerate the desired number of corrupt PGs.

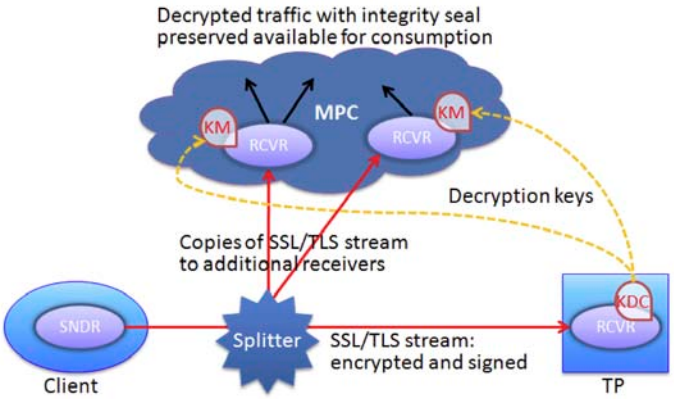
6. sIVPGs

At a high level, the function performed by the sIVPG is to a) replicate the encrypted stream without losing any application data, b) share keys so that the receiving end points (RCVRs) in the MPCs can decrypt and verify the integrity of the replicated SSL packets, and c) make the decrypted stream available to the MPs. We explored various implementation options including libpcap-based packet sniffers [4] to replicate the traffic stream, and settled on a netfilter-based approach [5] because the latter provides more robustness against packet loss.

In this approach, as soon as a client connection is initiated, the splitter component, as shown in Figure 4, starts to buffer traffic from that connection using a netfilter module. When the SSL handshake is completed and the PG in a MPC neighbor-

hood has been initialized to handle the new connection, the Key Distribution Center at the TP and Key Managers in the neighborhoods communicate to exchange the SSL keys. In parallel, the splitter starts to forward the buffered data to the RCVRs. The RCVRs buffer data until the key exchange step is completed, and make the decrypted data available through stream interface as soon as the necessary keys are available. Note that if the client-server messages are signed and encrypted at the application level, an additional level of key sharing is needed to make the decrypted data available for inspection and processing to the proxies.

Figure 4: sIVPGs



7. Recovery Focused Adaptation

The CZ is equipped with adaptation mechanisms that enable recovery and containment of attack effects. TPs and each MPC neighborhood have a watchdog that monitors the respective components and automatically restarts them when a crash is detected. A restarted component reconnects itself to its peers and begins handling new client connections. The watchdogs poll the components in a configurable interval (one second in our test configuration). Component restart time is dependent on configuration and load details. In our test configuration, components start in less than one second.

The CZ also maintains a database of log messages with database permissions set so that CZ components can write to the database (but not read) and only designated analysis components can read from the database (but not write). The logging mechanism collects data that will help the system prevent or minimize future attacks. For example, each time a check does something that might cause the neighborhood to crash (such as checking a serialized object through the canary proxy), it enters a log message. When it finishes executing the code that may cause a crash, it enters another log message. These log messages contain timestamps as well as the IP information about the connection under analysis.

The log analysis component analyzes the data collected in the log database. In particular it looks for indications that a particular client connection caused a crash. For example, a neighborhood that crashed might have a log message indicating that a block of potentially-crash-producing code was entered, but was never exited. The log analyzer can take proactive actions—either by modifying the firewall to prevent connections from a particular IP address or by assigning connections from an IP address to a high-risk neighborhood to further protect other client connections from potential crashes.

The watchdogs and logging ensure that the CZ remains available, is resilient to attacks, and is proactive in preventing or minimizing the effects of future attacks.

8. Related Work

Port Knocking [6] is similar to SPA, but SPA has the following advantages over Port Knocking: SPA is based on strong cryptographic ciphers, making spoofing more difficult, SPA packets are non-replayable, and SPA is robust against trivial sequence busting attacks.

SPA Implementations take different approaches; we explored two open-source implementations, Fwknop [7] and Knockknock [8]. These implementations differ in ways that might impact which one is chosen for a specific deployment. For packet encoding, Fwknop uses dedicated User Datagram Protocol packets while Knockknock encodes requests in TCP headers. This implies that Knockknock requires admin privileges on the client to generate customer TCP headers. For packet capturing, Fwknop uses libpcap (a large C library) to passively sniff SPA packets. Knockknock reads packet information from kern.log through a daemon that restricts root privileges to only ~15 lines of Python code. In our view, this makes the Knockknock daemon less likely to be subverted. Regarding functionality, Fwknop provides feature rich support for service ghosting and port randomization, while Knockknock follows a minimalistic approach.

Web Application Firewalls (WAFs) are designed to protect Java 2 Platform, Enterprise Edition applications and web services (WS) against common vulnerabilities listed in the Open Web Application Security Project top 10 list, e.g., Structured Query Language injection. While most WAFs are deployed at DMZ boundaries only and are hosted on hardened appliances, CZs are based on a lightweight distributed software paradigm that allows us to surround a selected set of services with fine-grained defenses. WAFs support only WS-related interaction models and lack support, for example, for other distributed protocols such as Java Remote Method Invocation (RMI).

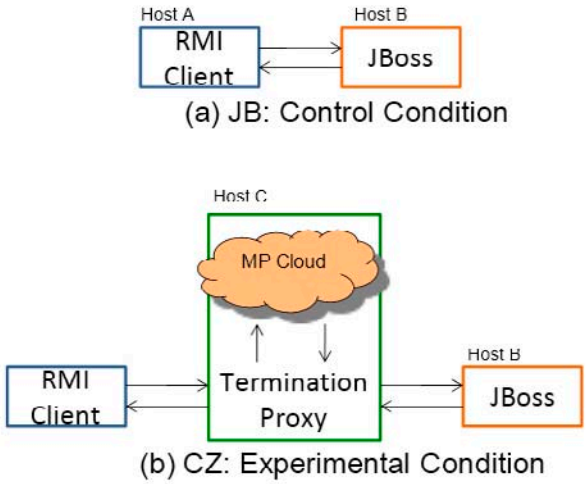
Application Server Clustering ensures availability of services by transparently rerouting traffic to redundant application servers in the presence of attacks that affect service availability. Load-balancers and clusters can work in conjunction with CZ to implement voting.

Extensible Markup Language (XML) Appliances provide security through schema validation, WS-security functions, and assured transformation of content using standards like Extensible Stylesheet Language Transformations. While there is some similarity between CZ MPs and functionality provided by XML appliances, XML appliances are based on a single hardened platform and don't provide advanced features such as canary proxies, diverse proxy implementation, and automatic restart.

Cross Domain Guards mitigate information exchange risks between different classified networks. New generation SOA-based guards [9][10] have started separating filter functionality into services that can be hosted outside of appliances, similar to the MPC. Compared to the work described here, existing certification and accreditation requirements play a more important role in guards, preventing the use of advanced techniques that don't fit current practices, e.g., use of canary proxies and probabilistic design algorithms.

9. Experimental Validation

Figure 5: Experiment Configurations



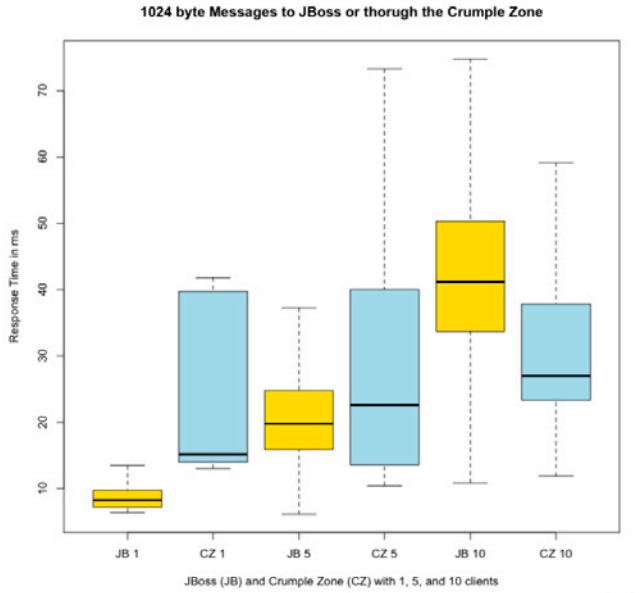
To evaluate the performance and robustness of the current proof-of-concept prototype CZ, we conducted multiple internal experiments. The system under test consisted of a Java RMI service and a MPC with four MPs, including rate, size, white list, and canary checks. Figure 5 shows the base and control conditions used.

We experimented with two categories of client messages: computation intensive and data intensive. The compute intensive messages are short but require the JBoss server to perform a mathematical calculation. The data messages were 1KiB, 10KiB, and 100KiB in size and required the JBoss server to process the data. As one might expect, the overhead of the CZ increases as the messages increase in size. This overhead ranges from 18% for the computation message to 84%, 803%, and 4040% for 1KiB, 10KiB, and 100KiB data messages respectively.

Our future work includes investigating and optimizing our code to handle large messages more efficiently. We suspect that the extra Input/Output (I/O) load analyzing the data could be responsible for the slower processing.

Additionally we investigated server response time when multiple clients make requests simultaneously. The results for testing one, five, and 10 clients are shown in the box plot in Figure 6. Interestingly, the response times for the CZ improve relative to the control condition as more clients are added. In fact, the median response time for the CZ is less than the median response time for the control condition when 10 clients connect at once. We believe that this improvement is due to the CZ sharing a connection to the JBoss server for all of the clients versus a separate connection to the JBoss server for each client in the control condition. This experiment shows that although there is overhead for a single client using the CZ this may be mitigated when multiple clients connect through the CZ.

Figure 6: Experiment results for multiple client connections



As shown in Figure 5, all of the CZ functions, including the termination proxy and the MPC, were hosted on a single host. While this configuration introduces minimal cost overhead in terms of additional hardware costs, I/O operations on the single machine will become a choke point given enough load. We plan to investigate other deployments in the future in which MPs are distributed across a set of machines in a load-balanced way. The expectation is that load-balanced configuration will decrease round trip times under heavy loads although increasing hosting costs.

10. Discussion and Next Steps

The CZ design and prototype described in this paper provides a promising foundation for protecting critical services from malicious attacks that succeed to a degree, i.e., get past the traditional access control and authentication services. This means that the CZ should be effective against novel, zero-day, and insider attacks.

The degree to which the CZ is effective against a particular attack depends greatly upon the extent to which the MPC replicates the server functionality and the kind of cross checking algorithms employed. In the extreme case, the Canary Proxy would replicate most of the server functionality, and would be susceptible to, and therefore provide protection against any attack that would be effective against the service, and the proxy group-TP processing would be made Byzantine Fault-Tolerant.

The amount of redundancy and protocol overhead must be weighed against the perceived threat model. One of the next steps that we are going to undertake is to evaluate the benefits and costs of protections and simulated functionality in the MPC, and how it fits particular threat models and platform performance requirements.

Similarly, the current prototype only protects the critical server from attacks by rogue clients. However, a fully protected system will want to protect the return path also, i.e., protect a client from a server that might have been compromised. To accomplish this, the return path from the client and server must go through a CZ. This CZ should be similar to the one on the request path, except that the functionality simulated by the Canary Proxy will involve processing of the response.

The current prototype concentrates on protecting server calls made over RMI. Although this is a valid model, representing calls made by composed clients and servers, a large class of client-server interactions in SOA are through WS interchanges, e.g., using Simple Object Access Protocol. We are currently in the process of designing a CZ that works with WS interfaces.

Finally, to substantiate our claims that the CZ can protect against large classes of known, zero-day, novel, and insider attacks, we plan to conduct experiments and collect concrete and empirical evidence. As we have done in prior research projects [11], we plan to conduct independent red team exercises to evaluate the efficacy of the CZ to protect against attacks by motivated and determined adversaries. In these exercises, an independent red team, experienced in cyber attacks and with insider knowledge of the system being protected, but not part of the development team, will launch attacks against the system. We will evaluate the ability of the CZ to absorb the attacks and protect the service, and the extent of the class of attacks that the CZ is effective against. To the extent possible, we will measure the difference in time to effectively compromise the system with and without CZ.

Acknowledgments

The authors would like to acknowledge the support and collaboration of the U.S. Air Force Research Laboratory (AFRL) Information Directorate. This material is based upon work supported by the Air Force Research Laboratory under Contract No. FA8750-09-C-0216.

ABOUT THE AUTHORS

Michael Atighetchi is a senior scientist at BBN's Information and Knowledge Technologies business unit. His research interests include cross-domain information sharing, security and survivability architectures, and middleware technologies. Mr. Atighetchi has published more than 35 technical papers in peer-reviewed journals and conferences, and is a senior member of the IEEE. He holds a master's degree in computer science from University of Massachusetts at Amherst, and a master's degree in IT from the University of Stuttgart, Germany.

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-1679
Fax: (617) 873-4328
E-mail: matighet@bbn.com

Dr. Partha Pal is a lead scientist at Raytheon BBN Technologies. He leads the survivability research thrust at Raytheon BBN, and has served as the principal investigator in a number of Defense Advanced Research Projects Agency (DARPA), Department of Homeland Security (DHS), and Air Force Research Laboratory Research & Development (AFRL R&D) projects in the areas of survivability and information assurance. He has published over 65 papers in refereed journals, conferences and workshops, has been in the program committees of multiple workshops and conferences, and has been a co-organizer of the Recent Advances in Intrusion Tolerance workshop for the past two years.

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-2056
Fax: (617) 873-4328
E-mail: ppal@bbn.com

ABOUT THE AUTHORS cont.

Dr. Aaron Adler is a Scientist in BBN's Information and Knowledge Technologies business unit. His research interests include distributed systems, artificial intelligence, and human computer interaction, specifically sketch recognition. He has a Ph.D. in Computer Science from Massachusetts Institute of Technology (2009).

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-3517
Fax: (617) 873-2794
E-mail: aadler@bbn.com

Andrew Gronosky is a staff engineer at Raytheon BBN Technologies. He has experience developing a variety of software applications including data analysis and visualization, digital signal processing, and parallel and distributed systems. He holds a Master of Science degree in mathematics from Rensselaer Polytechnic Institute and is a member of the IEEE and the Association for Computing Machinery.

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-3517
Fax: (617) 873-3486
E-mail: agronosk@bbn.com

Dr. Fusun Yaman is a Scientist in BBN Technologies. Her research interests are in distributed planning, spatio-temporal reasoning and machine learning specifically learning user preferences from observations. She has a Ph.D. in Computer Science from University of Maryland at College Park (2006).

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-3966
Fax: (617) 873-2794
E-mail: fyaman@bbn.com

Jonathan Webb is an engineer in BBN's Information and Knowledge Technologies business unit. Over 20 years at BBN, Mr. Webb has been involved in a wide range of software development projects including simulation of dynamic systems, web based data management systems, middleware for information management, and cross domain information sharing. Mr. Webb has a master's degree in aeronautics and astronautics from the Massachusetts Institute of Technology.

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-3321
Fax: (617) 873-4328
E-mail: jwebb@bbn.com

Dr. Joseph Loyall is a principal scientist at Raytheon BBN Technologies. He has been the principal investigator for Defense Advanced Research Projects Agency and AFRL research and development projects in the areas of information management, distributed middleware, adaptive applications, and quality of service. He is the author of over 75 published papers; was the program committee co-chair for the Distributed Objects and Applications conference (2002, 2005); and has been an invited speaker at several conferences and workshops. Dr. Loyall has a doctorate in computer science from the University of Illinois.

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138
Phone: (617) 873-4679
Fax: (617) 873-4328
E-mail: jloyall@bbn.com

Asher Sinclair is a Program Manager at AFRL's Information Directorate working in the Enterprise Information Management Branch at the Rome Research Site. His work history includes enterprise systems management, service-oriented architectures, information-level quality of service, and network security. He has contributed to more than 12 technical papers and conference proceeding publications. He holds a bachelor's degree in Computer Information Systems from the State University of New York and a master's degree in Information Management from Syracuse University.

AFRL
525 Brooks Road
Rome, NY 13441
Phone: (315) 330-1575
E-mail: asher.sinclair@rl.af.mil

ABOUT THE AUTHORS cont.

Charles Payne is a Member of the Technical Staff at Adventium Labs in Minneapolis, Minnesota. He has been a Principal Investigator for the Office of Naval Research in the area of virtualized cross domain support and has been a key contributor to DARPA, DHS and AFRL programs investigating high assurance security architectures. He has published more than a dozen papers and served as Program Chair for the Annual Computer Security Applications Conference (2009). Mr. Payne has a Masters of Science degree in Computer Science from The College of William and Mary in Virginia.

Adventium Labs
111 Third Avenue South, Suite 100
Minneapolis, MN 55401
Phone: (612) 817 2525
E-mail: charles.payne@adventiumlabs.org

REFERENCES

1. Michael Atighetchi, Partha Pal Andrew Gronosky, "Understanding the Vulnerabilities of a SOA Platform - A Case Study," in The 9th IEEE International Symposium on Network Computing and Applications (IEEE NCA10) , Cambridge, MA USA, 2010.
2. Eric Rescorla, SSL and TLS: Designing and Building Secure Systems. United States: Addison-Wesley Pub Co., 2001.
3. Michael Rash. (2006) Single Packet Authorization with Fwknop. [Online]. <http://www.cipherdyne.org/fwknop/docs/SPA.html>
4. (2010, September) TCPDUMP home page. [Online]. <http://www.tcpdump.org/>
5. (2010, September) Netfilter Homepage. [Online]. <http://www.netfilter.org/>
6. Martin Krzywinski. (2005) portknocking.org. [Online]. <http://www.portknocking.org/docs/portknocking_an_introduction.pdf>
7. CipherDyne. (2010, September) CipherDyne. [Online]. <http://www.cipherdyne.org/fwknop/>
8. Moxie Marlinspike. (2010, September) KnockKnock. [Online]. <http://www.thoughtcrime.org/software/knockknock/>
9. Jason Ostermann. (2009) Presentation at UCDOM Annual Conference: Raytheon DSCDS Intro. [Online]. <http://www.ucdmo.gov/conference09/Ostermann_Raytheon%20DSCDS_09022009.pdf>
10. BAH. (2009) Presentation at the UCDMO Annual Conference: BAH DSCDS Overview. [Online]. <http://www.ucdmo.gov/conference09/Morris_BAH%20DSCDSoverview_final_09022009.pdf>
11. Joe Loyall Michael Atighetchi, "Meaningful and Flexible Survivability Assessments: Approach and Practice," in CrossTalk - The Journal Of Defense Software Engineering, March/April 2010, pp. 12-18.

Stop the Flow

A Damage Mitigation Framework for Trusted Systems

Linda M. Laird, Stevens Institute of Technology
Jon P. Wade, Stevens Institute of Technology

Abstract. This article proposes a very high-level, abstract framework of Damage Mitigation to enable the architecture and design of trusted systems, those that dependably perform a mission while minimizing or eliminating the probability of significant, unintended damage. This framework is based upon the premise of system imperfection, consisting of a Trusted Systems Model and a Damage Process Model. The intent is that this systems approach to Damage Mitigation will improve the ability to analyze the Damage Mitigation capabilities of a system and encourage new solutions.

Introduction

A Trusted System is one that dependably performs its mission while minimizing or eliminating the probability of significant, unintended damage. The ability to develop, deploy, and maintain trusted systems, those that are safe, secure, dependable, and survivable, is an unsolved problem. There are calls for papers looking for new approaches to address these issues [1] [2] as radiation systems continue to occasionally kill people [3], cars occasionally refuse to stop, and rogue botnets are available for hire [4]. If anything, our experiences in the last 30 years of building software intensive systems have shown that software and systems without defects or vulnerabilities are, for all practical purposes, non-existent. This is not to say that the utmost should not be done to eliminate defects and vulnerabilities. It must. But time and effort are limited, as are human abilities and knowledge, while system complexities continue to increase. We must assume that defects and vulnerabilities will always exist.

This article proposes a very high-level, abstract framework of Damage Mitigation based upon this premise of imperfection. Damage will be defined as any significant negative consequence of a system's operation. The intent is that this systems approach to Damage Mitigation will facilitate new ideas on how to improve the fundamental properties of trustability of systems and encourage the creation of trustable architectures and designs for critical systems. The central idea is that there exists a causal event chain that can lead to damage and a loss in system value. At each point within the chain, there are potential "chokepoints" where it may be possible to stop the flow from an instigating event to a damage event.

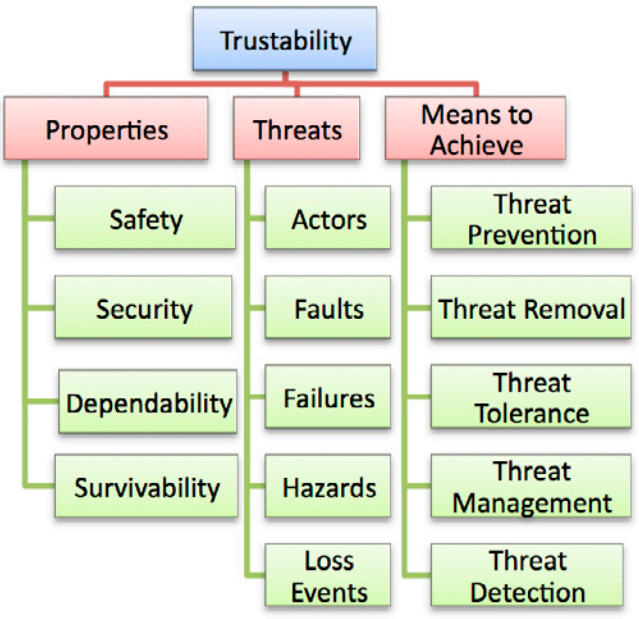
The broader framework and the different viewpoint of this article hopefully will encourage and suggest new solutions to the problem of building, engineering, and operating systems that need to be trusted. Alan Kay once remarked, "A change in perspective is worth 80 IQ points." The experience to date has been that in using this framework, additional solutions and mitigation strategies to past accidents became apparent.

System Trustability Model

The Trustability Model shown in Figure 1 has three attributes: Trustability Properties, Threats to Trustability, and Means to Achieve Trustability. It is based upon the Dependability Model in [5]. This model shows that there is a strong relationship between the various properties of trustability, the threats to those properties, and the means to achieve those properties. For example, Threat Prevention has a positive impact on all of the properties of trustability, with perhaps the exception of resilience. Threat Tolerance has a positive impact on all of the properties. Being able to tolerate and eventually remove loss events improves resilience, availability, and security.

In this model, safety reflects the system's ability to protect its users and environment from both physical and non-physical threats. Security describes the ability of the system to protect itself with respect to the confidentiality of its assets and its overall integrity. Dependability ensures that the system provides its services and supports its mission when it is required. The properties of availability, reliability, and maintainability are all elements which constitute dependability. Finally, survivability is a measure of the system's capability to support the attributes noted above despite adverse environmental effects. The properties of robustness and resilience contribute to this capability.

Figure 1: Trustability Model



System safety, security, dependability and survivability are proposed as the critical properties of trustability. The relative importance and need for each individual property differ by domain, application, and context of the system. The definitions of these terms and those for some of their constituent properties that will be used in this work are described below.

Safety is the ability of a system to perform predictably under normal and abnormal conditions, thereby preventing accidental injury, death, or damage. This definition has been adapted [6] with a key change being the addition of the damage concept. Although safety, in general use, typically has the connotation of physical safety, with software intensive systems, safety is the antithesis of dangerous and can relate to non-physical safety as well.

Security is the ability of a system to thwart attempts to exploit defects and vulnerabilities to disrupt the system or the value it produces. (Adapted from [7].) Security adds the concepts of perpetrators and malicious attackers.

Dependability is "the ability to deliver service that can justifiably be trusted." [5] The dependability of a system is based upon its reliability, availability, and maintainability. The relative importance for each of these is determined by the context in which the system is being used.

Survivability is the ability of a system to function during and after a natural or man-made disturbance. (Adapted from [8].) For a given application, survivability must be qualified by specifying the range of conditions which the system will survive along with the minimum acceptable levels of safety, security, and dependability. Resilience and robustness are two important properties which determine survivability.

Threats to Trusted Systems

A Threat to a Trusted System is anything that can potentially cause the system to have significant unintended damage, including not being able to complete its mission successfully. This definition includes both malicious and unintended threats. In this model, the set of threats includes actors, faults, failures, hazards, and loss events.

The term Actor is borrowed from Unified Modeling Language (UML) and well suited to this use. Within this context, an Actor is anything that instigates execution of the system. Actors include humans, systems (external and internal), the physical world, and any other external object that can act upon the system. Actors instigate execution of the system, and in doing so provide inputs, friendly and malicious, planned and unexpected, that can cause failures, hazards, and ultimately, significant damage.

A fault is a defect or vulnerability in the system that may or may not cause a failure. A fault might be a memory leak that can lead to a system crash or incorrect execution. An exploitable buffer-overflow vulnerability is a fault. A requirements defect is a fault. For software, if the code that contains the faults is never executed, or never executed under the precise conditions that cause the fault to occur, then the system never fails due to this fault. Faults are defects in the system that may or may never be seen in operation.

A failure occurs when the object (component, human, system) can no longer create value (carry out its mission) or no longer delivers the correct service. Failures only occur during system execution.

A hazard is a state or set of conditions of a system that, together with other conditions in the environment of the system, will probabilistically lead to a loss event, be it an accident or incident. A hazard represents the possibility of a loss event. Hazards have the following attributes: severity, likelihood of occurrence, exposure (duration), and likelihood of a hazard leading to an accident.

A loss event, within the context of this work, is an accident, incident, or other unsuccessful completion of the mission of the system. Loss events vary in significance. An incident is considered to be a loss event that involves minor or no loss but with the potential for significant loss under different circumstances. A loss event can be mitigated to minimize damage.

Means to Achieve Trusted Systems

Trustability is achieved when loss events either do not occur or the unintended damage caused by them to the stakeholders is deemed to be acceptable. In order to achieve trustability, the threats to trustability need to be mitigated. There are a significant number of mitigation techniques categorized below. While there have been numerous attempts at this categorization, this work will use the following definitions:

1. Threat Prevention: the set of techniques to assure that the threat is not allowed into the system.
2. Threat Removal: the set of techniques to remove threats from the system.
3. Threat Tolerance: the set of techniques that prevent a threat from causing damage.
4. Threat Management: the set of techniques to minimize the potential damage.
5. Threat Detection: the set of techniques that allow threats to be observed. Threat detection could be included as an enabler to all of the other Means to Achieve. In this work, it is noted separately as it is required to demonstrate that the system is trustworthy. Monitoring allows the current threat patterns to be understood and potentially supports the prediction of future threat patterns.

All of these techniques have advantages, although threat prevention is obviously preferred. If malicious actors never contact the system, or vulnerabilities never exist, the damage that they might cause is totally mitigated. When threats cannot be prevented or removed, threat tolerance, which stops the propagation of potential damage, is useful. Threat tolerance includes basic techniques such as rollback and recovery. Threat management can be used once a threat condition is recognized and until the root cause can be addressed. For example, quarantining off parts of a system that have been compromised is a threat management technique.

Damage Process Model

A casual model of the Damage Process is shown in Figure 2, consistent with the definitions above. It is a set of steps, starting with an actor, and terminating with damage. If a fault is encountered, and it causes a failure, it can create a hazardous situation that under certain circumstances can cause a loss event, which, if not properly mitigated can cause unacceptable damage. It is important to emphasize that this damage process is one

Figure 2: Damage Process Steps

scenario of a system-in-the-large in execution. No damage occurs unless the process is initiated. In some respects, it could be considered a high-level scenario or use case.

As an example, consider the St. Vincent's radiation tragedy [3], in which at least two patients were killed by incorrectly applied radiation dosage, apparently due, in part, to defects in the fault management software and system design. "… as (the technician) was trying to save her work, the computer began seizing up, displaying an error message. An error message asked (the technician) if she wanted to save her changes before the program aborted. She answered yes. … (the Doctor) approved the new plan." Two rounds of radiation treatment were given. The technician ran a test to verify the settings before the third round. "What she saw was horrifying: the multileaf collimator, which was supposed to focus the beam precisely on his tumor, was wide open." [3]

The patient subsequently died from the incorrect and excessive radiation. The equipment provider subsequently "distributed new software, with a fail-safe provision." [3]

Each step leading up to the final damage result is considered a potential threat to the trustability of the system. At each step, there is the possibility of mitigating the threat to prevent or minimize the final damage. There are possible intervention points, or chokepoints, after each step of the process model, as shown below in Figure 3.

At each chokepoint, techniques could be applied to both monitor and mitigate potential threat conditions. Using this radiation example, the flow to the significant damage event could have been stopped if:

a. The technician did not save her work when the system was crashing. Additional training or documentation may have worked.

b. The system had improved fault handling that either stored the work correctly or did not allow potentially compromised configurations to be stored.

c. The display of the information to the doctor, who had to sign off on it, may have been complex and difficult to verify. A visual display/simulation may have improved it significantly.

d. The technician was required to run a test to verify the settings before first use (which she did before the third use).

e. The system had safety checks (either internal to the algorithms or even on the radiation devices) that prevented extremely high dosage radiation or forced the operator to repeatedly confirm that this high level was intended.

f. The overall system was designed to first test the radiation settings on a radiation "dummy," with safe dosage confirmation before continuing with a live patient.

g. There were new medicines that could reverse the effects of over-radiation.

This is an example of different solutions that are relatively easy to uncover by considering the damage event as a process with many possible intervention points, rather than identifying the problem as a bug in the code. These solutions are examples of possible "safe(r) designs." From a systems viewpoint, solution f is especially interesting because it stops incorrect radiation in almost all cases, no matter the cause. It would "Stop the Flow" before any loss event could occur.

Discussion of Mitigation Techniques

It is important to realize that the mitigation techniques of focus here are for the execution process, not the development process. Preventing faults and vulnerabilities from entering the system during development is critically important and is a relatively well developed field, but in this model, we assume that regardless of the quality of the developers and the development process, faults and vulnerabilities will exist.

To illustrate the ease of uncovering mitigation techniques, examples of generic techniques for each chokepoint are listed. These generic techniques could be easily expanded and used as a brainstorming tool or checklist during the design of critical systems.

The first chokepoint focuses on actors and fault prevention. Some possible issues might arise from unauthorized users, incorrect usage, or system overload. Example mitigation techniques include denying access to unauthorized users of the system, denying access to "untrusted" external systems, deliberate shedding of users in overload conditions, and input and interface validation.

At the second chokepoint, typical examples would be traditional

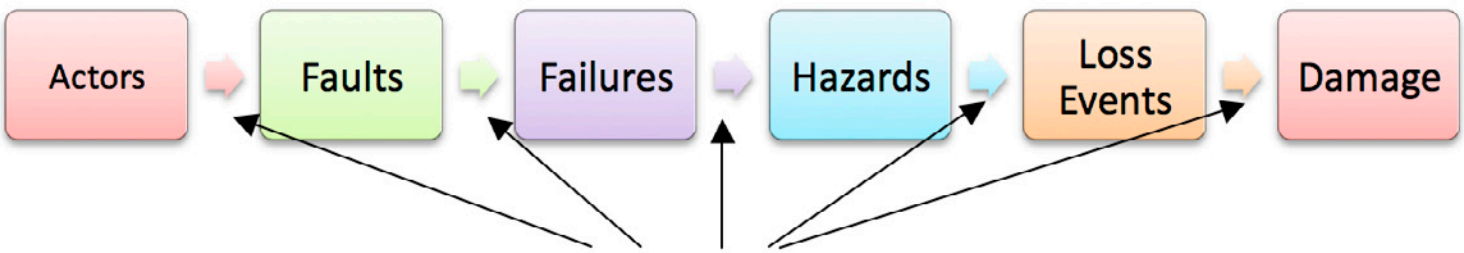


Figure 3: Chokepoints

techniques such as robust fault handling, fault tolerance, fault tolerant middleware and software rejuvenation, or the use of error correction codes in hardware storage or communication systems.

At the third chokepoint, between failures and hazards, typical techniques would include failure management capabilities, such as recovery, redundancy, an overarching failure management strategy, as well as fail-safe and fail-operational capabilities.

At the fourth chokepoint, once a hazardous state occurs it may still be possible to recognize that a hazardous condition exists, and prevent an accident by implementing accident avoidance capabilities. Techniques for survivability, including system resilience and the concept of degraded modes of operation, such as in [9] and [10] are likely to be applicable at this chokepoint as well.

At the fifth chokepoint, even after a loss-event occurs, there are mitigation techniques that can be applied. For example in an automobile, if an accident occurs, air bags and guard rails are mitigation techniques to minimize the resultant damage. Examples of generic techniques are more difficult to define at this chokepoint; they seem to be more system dependent, although analysis techniques such as Failure Modes and Effects Analysis are applicable.

The damage mitigation techniques for software-intensive systems naturally have been focused more at the beginning of the damage process with less emphasis towards the end. Much is known and published about fault avoidance and fault tolerance. Less is known about failure management. Little has been written about the recognition and management of hazardous conditions in software-intensive systems. Loss-event avoidance appears to be relatively unexplored, with the exception of Sha's work on a Simplex Architecture [11], [12] on safety-critical systems, and the previously mentioned work on defining survivable core capabilities [10] and [9].

Conclusion and Future Work

The objective of this article is to provide a conceptual framework for damage mitigation and trusted systems to enable the research, design, and operation of mission-critical systems. While it has been used successfully in a graduate level software engineering course, this work is still in an embryonic stage. Future work includes:

- Conducting case studies of actual mission-critical systems to determine the utility of the proposed framework, the classification of existing damage mitigation techniques and architectures, and exploring novel techniques which the framework suggests.
- Analyzing techniques that may be used to recognize hazardous conditions during runtime as a means to deploy damage mitigation.
- Investigating the use of control system theory as a means to monitor and control the damage mitigation process.
- Researching the relationships between the attributes of safety, security, dependability, and survivability.

REFERENCES

1. "DSN 2010." Dependable Systems Conference 28 July 2010. < http://www.dsn.org/>
2. "HICSS." Hawaii International Conference of System Science 4 Jan. 2011. <http://www.hicss.hawaii.edu/>
3. Bogdanich, Walt. "Radiation Offers New Cures, and Ways to Do Harm." The New York Times 24 Jan. 2010. <http://www.nytimes.com/2010/01/24/health/24radiation.html>
4. Wilson, Clay. Botnets, Cybercrime, and Cyberterrorism: Vulnerabilities and Policy Issues for Congress. 17 Nov. 2007. <http://www.fas.org/sgp/crs/terror/RL32114.pdf>
5. Avizienis, A. et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing." IEEE Transactions on Dependable and Secure Computing 1.1, (Jan-Mar 2004): 11-33
6. Herrmann, Debra S. Software Safety and Reliability: Techniques, Approaches, and Standards of Key Industrial Sectors. Los Alamitos, California: IEEE Computer Society, 1999: 14-15
7. Bayuk, Jennifer. "The Utility of Security Standards." Washington, D. C., 2010.
8. Federal Standard 1037C. "Federal Standard Telecommunications: Glossary of Telecommunications Terms." 1996. <http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm>
9. Knight, John C, and Kevin J Sullivan. "On the Definition of Survivability." University of Virginia, Department of Computer Science Technical Report CS-TR-33-00, 2000. <http://www.cs.virginia.edu/~jck/publications/tech.report.2000.33.pdf>
10. Linger, Richard, Nancy Mead, and Howard Lipson. "Requirements Definition for Survivable Network Systems." Proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice, 1998. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.6083&rep=rep1&type=pdf>
11. Sha, Lui. "Using Simplicity to Control Complexity." IEEE SOFTWARE (July/Aug 2001): 20-28 <https://agora.cs.illinois.edu/download/attachments/10581/IEEESoftware.pdf?version=1>
12. Holzmann, G. "Software Safety and Rocket Science." ECRIM News, Special Safety-Critical Software, 75 2008. <http://ercim-news.ercim.eu/software-safety-and-rocket-science>

ABOUT THE AUTHORS

Linda Laird is the Director of the Software Engineering Program at Stevens Institute of Technology. Professor Laird is the author of Software Estimation and Measurement: A Practical Approach. Her research interests include system dependability, architecture and design for software reliability, and software estimation. She is a graduate of Carnegie-Mellon University and the University of Michigan. Professor Laird managed large software development projects at Bell Laboratories for over 25 years.

Jon Wade, Ph.D. is the Associate Dean of Research at the School of Systems and Enterprises at the Stevens Institute of Technology. Dr. Wade's research interests include the transformation of systems engineering, Enterprise Systems and Systems of Systems, and the use of technology in technical workforce development. He has over 20 years of experience in the research and development of Enterprise systems at IGT, Sun Microsystems and Thinking Machines Corporation. Dr. Wade is a graduate of the Massachusetts Institute of Technology.

Linda M. Laird, Jon P. Wade
Stevens Institute of Technology
School of Systems and Enterprises
Castle Point on Hudson
Hoboken, NJ 07030

The Need for Functional Security Testing

C. Warren Axelrod, Ph.D., Delta Risk

Abstract. Despite extensive testing of application functionality and security, we see many instances of software, when attacked or during normal operation, performing adversely in ways that were not anticipated. In large part, this is due to software assurance staff not testing fully for "negative functionality," that is, ensuring that applications do not do what they are not supposed to. There are many reasons for this, including the relative enormity of the task, the pressure to implement quickly, and the lack of qualified testers. In this article, we will examine these issues and suggest ways in which we can achieve some measure of assurance that applications will not behave inappropriately under a broad range of conditions.

Introduction

Traditionally, software testing has focused on making sure systems satisfy requirements. Such functional requirements and specifications are expected to, but may not necessarily, accurately depict the functionality actually wanted by prospective users, particularly those aspects users may not be aware of or may not have been asked to consider.

In this article we examine the issues and challenges related to ensuring applications do not do what applications are not supposed to do. Such testing, for which we use the term Functional Security Testing (FST), is often complex, extensive and open-ended. And yet it is key to the secure and resilient operation of applications for the applications not to misbehave when subjected to various adverse stimuli or attacks.

The Evolution Of Testing

Programmers test applications that they are developing to ensure the applications run through to completion without generating errors. Programmers then usually engage in some rudimentary tests for correctness, such as ensuring that calculations correctly handle the types of data the programs process. In general, programmers seldom think "out of the box." This attribute was, to a large extent, the root cause of the Y2K "bug," where programmers frequently did not anticipate their programs would be running after Dec. 31, 1999 and so did not include century indicators in the programs, opting for two-digit year depictions. While it is true that programmers were motivated to abbreviate the year field by the need to stay within strict limitations in the amount of data that could be stored and transmitted, they failed to look to the future when their programs would crash.

During this early period, programs were thrown "over the transom" to the Quality Assurance or Software Assurance departments, where test engineers would attempt to match the functioning of the programs against the functional specifications developed by systems analysts. In general, such testers would limit their scope to ensuring the programs did what was intended, and not consider anomalous program behavior.[1]

Over the past decade, there has been greater focus on what might be called technical security testing, where security includes confidentiality, integrity, and availability. The usual approach is to assess the adherence of systems (including applications, system software and hardware, networks, etc.) to secure architecture, design, coding (i.e., programming), and operational standards. Often such testing includes checking for common vulnerabilities and programming errors, such as those specified by the Open Web Application Security Project (OWASP)[2] and SysAdmin, Audit, Network, Security (SANS) [3] respectively.

However, there are aspects of security testing that are different. For example, McGraw [1] refers to "anti-requirements," which "provide insight into how a malicious user, attacker … can abuse [a] system." McGraw differentiates anti-requirements from "security requirements" in that the security requirements "result in functionality that is built into a system to establish accepted behavior, [whereas] anti-requirements are established to determine what happens when this functionality goes away." McGraw goes on to say "anti-requirements are often tied up in the lack of or failure of a security function." Note that McGraw is referring to the adequacy or resiliency of security functions and not functions within applications.

Merkow and Lakshmikanth [2] refer to security-related and resiliency-related testing as "nonfunctional requirements (NFR) testing." NFR testing, which is used to determine the quality, security, and resiliency aspects of software, is based on the belief that nonfunctional requirements represent not what software is meant to do but how the software might do it. Merkow and Lakshmikanth [2] also stated that "gaining confidence that a system does not do what it's not supposed to do …" requires subjecting "… a system to brutal resilience testing."

In his book [3], Anderson affirms the importance of resilience testing with his comment that: "Failure recovery is often the most important aspect of security engineering, yet it is one of the most neglected." He goes on to note that "… secure distributed systems tend to have been discussed as a side issue by experts on communications protocols and operating systems …"

The author believes FST is another key area of testing that has received little attention from application development and information security communities and is not specifically mentioned in [1] or [2] or other publications.[4] Using FST, applications are tested to ensure they do not allow harmful functional responses, which might have been initiated by legitimate or fraudulent users, to take place. It should be noted here that testing for responses that do not derive specifically from functions within applications, such as when a computer process corrupts data, are not included in the author's definition of FST (see Introduction section).

It is important to differentiate among functional testing[5] of applications, which attempts to ensure that the functionality

of applications matches requirements; security testing, which aims to eliminate the aspects of systems that do not relate to application functionality but to the confidentiality, integrity, and availability of applications and the applications' infrastructure; and FST, which is designed to ferret out the malfunctioning of applications that might lead to security compromises.

In this article, we examine FST and how it relates to other forms of testing, look at why it might have received so little attention to date, and suggest what is needed to make it a more effective software assurance tool.

Categories Of Testing

Various types of testing are key for successful software development and operation, as discussed in [1], [2], and [3]. As described previously, software testers (or test engineers) most commonly check that computer programs operate in accordance with the design of an application and consequent functional specifications, which in turn are meant to reflect users' functional requirements. This form of testing is termed "functional *requirements* testing." When applications are tested for functionality in isolation, rather than in an operational context, the activity is called "unit testing." However, testers do need to ensure applications work correctly with the other applications with which they must interact. This latter form of functional requirements testing is known as "integration testing." And testers must further check that individual programs function appropriately in the various particular contexts to which they might be subjected. This further form of functional requirements testing is known as "regression testing," which is done to assure that changes in the functionality of an application do not have a negative impact on other components, subsystems and systems.

There is increasing interest from information security and risk management professionals in the security strength of software, as shown by the growth of such organizations as OWASP at <http://www.owasp.org>, which has seen very rapid growth in membership since its founding in 2001, and the Build Security In collaborative effort sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security, at <https://buildsecurityin.us-cert.gov/bsi/home.html>. Such interest was largely precipitated by the discovery of a growing number of highly effective attacks against the application layer. Some estimate that such attacks account for as much as 70% of attacks.[6] As a result, we are seeing considerable growth in security testing of applications, as indicated by the increasing activity of organizations such as OWASP, mentioned above, as well as a strong movement for building security into applications from the beginning.[7] Security testing is essentially geared to reviewing software designs and coding practices, and the software code itself, to ensure the most egregious vulnerabilities have not been introduced into the concept, design, architecture, specifications, requirements, building, or release phases. When such exposures are discovered, they must be quickly eliminated. Security testing is essentially negative testing in that testers try to determine that certain attacks will not be successful. If the attacks are deemed to be a threat to a specific piece of software, then testers recommend they be eliminated.

While functional testing is commonplace, and security testing is increasingly popular, there is a third form of testing, which we term FST, which is not generally applied. This circumstance might be ignored were it not for the fact that this latter form of testing can be one of the most important aspects of software assurance. This is because it is this form of testing that provides some level of assurance that applications will not allow activities that should not be permitted. The desired assurance level depends on the cost and time (or any resulting delays) of the FST exercise and the benefits derived in terms of avoidance of malfunctions and resistance to attacks.

Some Examples Of FST

Perhaps the most ubiquitous example of the need for FST occurred during the Y2K remediation period. As the reader will recall, many computer programs, particularly within legacy systems, used two digit year data fields and did not recognize the date change to the 21st century. This might be considered a security issue as the integrity and availability of applications were at risk due to the Y2K problem, and the confidentiality of information was threatened in some situations. As a result, estimated expenditures of more than $300 billion were incurred to modify programs so they would correctly handle the millennium date rollover.[8] Despite these efforts, a large number of programs retained this defect and failed to process dates correctly, although catastrophic failure was avoided for the year 2000. For the most part, testing for the Y2K "bug" was a form of FST in that assessment of programs was on the basis of failure, not positive functionality, as previously mentioned.

More recently, there was a report of the interception of video transmissions from drones on surveillance missions in Iraq and Afghanistan where enemies could easily view video feeds using a $26 piece of commercially available software [9]. This is another example of systems that permit misuse because they were not tested against specific eventualities.

Increased Security Testing

About a dozen years ago, as the chief information security officer of a financial services firm, the author was asked to develop a series of scripts for testing the security of a strategic new client-server system. The testers had already created some 600 functional test scripts. The author came up with some 10,000 scripts for testing the "security" of the system. The number was large because there were so many possible combinations and permutations of ways to access functions and data. In the author's opinion, this ratio of testing scripts is typical.

During the intervening years, there has been increased interest in and growth of technical security testing, in the author's opinion, as a means of assuring the security of applications and systems. Such testing is critical to the secure operation of applications and does much to reduce vulnerabilities to attacks through the application layer. The range of such testing is very well described in several books, especially in [1]. However, this type of testing is performed by security testers who are familiar with the security weaknesses of programming languages and ways in which designers and programmers introduce vulnerabilities. These practitioners seldom get into the functionality of the applications and what security (i.e., confidentiality, integrity, and availability) exposures there might be in the functional operation of the software.

Skills Required Of Test Staff

The evolution of testing, from functional testing to security testing to FST, clearly relates to particular threats and the skill pools available at the time. In the early days of software development, the primary goal was to ensure the program worked and performed according to the requirements, with any residual errors showing up and being fixed during operations. Early mainframe and minicomputer systems were usually accessed and used by insiders or outsiders under the supervision of insiders. Programmers tested for the failure-free running of the software, whereas the testing staff ascertained that the functionality of the software was correct. Functional errors were turned back to the developers for correction and the software was retested until it operated correctly. In the author's experience, better-qualified test staffs were often familiar with the business use of the applications and had some rudimentary understanding of programming and computer operations.

With the arrival and proliferation of the Internet, applications were increasingly accessed and used by outside parties. This made for the need to test for a greater degree of security since organizations operating the systems often do not have much control over the actions of end users. The same deterrents that can be used against internal miscreants for their misuse of a system generally are not particularly effective against customers, business partners, or the public at large, since the latter are not subject to the same consequences, such as termination of employment.[9] Consequently, it has fallen to software vendors and internal staff to attempt to make sure that the systems cannot be compromised by evildoers both inside and external to the organization operating the software.

Table 1 shows the levels of knowledge, skill, and experience needed for each type of testing as well as how such attributes vary with the type of testing being conducted.

FST Issues

One of the issues relating to FST is that it does not yet have a generally accepted, immediately recognizable name. The type of testing referenced here lies somewhere between traditional functional testing and security testing, as we currently know these test categories. As described above, the former is used to establish that applications operate according to the functionality requirements expressed in the design of the system, that is, they do what they are supposed to do. The latter is a combination of tests relating to the security quality of the design, architecture, and coding aspects of an application, as well as other characteristics pertaining to the platform on which an application runs and the infrastructure that supports the system.

Other factors, such as the context in which the system will operate (e.g., Web facing, open, and internal), have a major impact on the level of testing that should be performed in each category. Criti-

Table 1: Required Knowledge, Skills, and Experience for Different Testing Approaches

Knowledge Skills and Experience Requirements for	Functional Requirements Testing*	Nonfunctional Requirements (Security) Testing**	Anti-Requirements Testing***	FST****
General business	Moderate	Low	Low	Moderate
Business processing	Moderate – High	Moderate – Low	Low	High
Coding standards	Moderate – Functional coding standards	High – Security coding standards	High – Security coding standards	High – Functional and security coding standards
Testing Methods	High – Functional testing	High – Security testing	High – Security testing	High – Functional and security testing
Computer operations	Moderate	High	High	High
Security – Privacy and Confidentiality	Moderate - High	High	High	High
Security – Integrity	Low to Moderate	High	High	High
Security – Availability	Low	High	High	High

* *Per McGraw [1], Merkow and Lakshmikanth [2] and Anderson [3]*
** *Per Merkow and Lakshmikanth [2]*
*** *Per McGraw [1]*
**** *Per this article*

cal Web-facing commercial applications and embedded systems operating aircraft or weaponry must undergo extensive testing.

Between these two traditional test modes, are more recent approaches. One is called "anti-requirements testing" or "negative requirements testing"[10] of applications. This is directed at the security aspects of systems and its purpose is to prevent security-related components from behaving badly.

The fourth category is FST, which is essentially testing to ensure the application does not allow application functionality and data use that is forbidden, either implicitly or explicitly, which might compromise security. An example of such a test objective would be … "Do not allow one customer to see another customer's data." To test this fully, every possible combination of user access to functions and data, both authorized and unauthorized, must be tested, which is impossible in practice. Therefore, some compromises must be made as to how much of this testing can reasonably be done, subject to project time and funding constraints.

Why So Little Attention To FST?

The author believes that there are a number of reasons why there is insufficient emphasis on FST, namely:

- FST can be orders of magnitude greater in effort and cost than traditional application functional testing and security testing.
- To perform FST properly, testers must be knowledgeable and experienced in business functions and application security, as well as software assurance processes.[11]
- The importance of testing for negative functionality is, in the view of the author, not generally recognized by general business management, risk managers, IT management, and application development managers.

What Needs To Be Done?

Support from Owners and Participants:

The most important step is to gain support for FST and get various participants and owners to understand that there is a very significant gap in standard software testing. This gap shows up when, for example, insiders are able to access information to which they should not have access, and then use the information for nefarious purposes.

FST Scope and Procedure:

The scope, procedure, time frame, and cost for a particular FST exercise have to be determined in advance. In many cases, the cost of running through all possible FST scripts is prohibitive and cannot be justified economically. Therefore it is necessary to create an iterative, adaptive procedure. One such approach is to test random samples from the entirety of test scripts and determine, using statistical methods applied to the design of experiments, from the results of the sample tests whether it is worthwhile to test further samples. In the author's experience, this approach results in running a relatively small subset of the complete list of test scripts, which usually reduces the cost and duration of testing considerably, while improving the accuracy of the results.

The author recommends an FST methodology which consists of the following steps:

1. Review test scripts that have been created for functional requirements testing.
2. Create as complete a set as possible of potential user activities (use cases) and remove those activities that are included in the functional requirements test scripts.
3. Develop test scripts for the remaining activities.
4. If the number and size of test scripts of the remaining activities are too many and too large, respectively, to justify the expense and time for such testing as determined by a return on investment (ROI) analysis, adopt a method for testing a succession of random samples of the scripts (where the size of the sample is based on ROI) and run the selected scripts.
5. If the error rates are significant then the sample size should be expanded based on the risk suggested by the prior tests.
6. The application and system errors, which are detected in the FST process, are fed back to the development team for correction and the corrected code is then retested until all material errors are fixed.

ROI:

Regarding ROI, the cost of and time required for performing tests can generally be estimated with a reasonable degree of accuracy. However, with FST and NFR (security) testing, the dependency of future sample sizes on the results of prior tests makes for dynamic costs and durations. In addition, the benefits of FST and NFR testing are particularly difficult to determine since, when sampling is involved, it is not clear what errors might remain.

In practical terms, FST should continue until the development and testing teams are reasonably satisfied that the applications no longer harbor any major latent deficiencies, subject to maintaining a positive ROI from the FST process itself.

Body of Knowledge:

Finally, we need to develop a body of knowledge about FST from experience with successful tests and actual software failures. As the library of such tests expands, it is important to share test results with others who can then apply lessons learned to their own FST programs. In this way, testers can more readily determine which types of tests are likely to be more fruitful and these testers, in their turn, can contribute new FST facts and experiences to those already cataloged.

Summary and Conclusions

The testing of applications to try to ensure that they do not misbehave is complex, sophisticated, and expensive. Yet the cost of not doing such tests, in terms of security incidents, can be so much greater than going through a realistic FST exercise.

There may well be some centers where this type of testing is already being performed, but is not called FST. However, the author believes it is safe to say that FST is not ubiquitous, as can be seen from the flood of incident information that often appears in the news.[12]

It is also apparent from the lack of published material in this area that developers and security professionals are not generally familiar with the principles of FST and therefore do not practice them to the detriment of system confidentiality, integrity, and availability. The situation will only improve when it is generally accepted that we need to ensure applications are prevented from functionally allowing damaging events. This is in addition to the non-functional security testing that is more commonplace.

ABOUT THE AUTHOR

C. Warren Axelrod, Ph.D., is a senior consultant with Delta Risk, a consultancy specializing in cyber defense, resiliency and risk management. Previously, he was the chief privacy officer and business information security officer for US Trust, the private wealth management division of Bank of America. He was a co-founder of the Financial Services Information Sharing and Analysis Center. Dr. Axelrod won the 2009 Michael Cangemi Best Book/Best Article Award for his article "Accounting for Value and Uncertainty in Security Metrics," published in the Information Systems Audit and Control Association (ISACA) Journal, Volume 6, 2008. He was honored with the prestigious Information Security Executive Luminary Leadership Award in 2007. He received a Computerworld Premier 100 IT Leaders Award in 2003.

Dr. Axelrod has written three books, two on computer management, and numerous articles on information technology and information security topics. His third book is *Outsourcing Information Security*, published in 2004 by Artech House. His article "Investing in Software Resiliency" appeared in the September/October 2009 issue of CrossTalk magazine.

He holds a Ph.D. in managerial economics from Cornell University, as well as an honors M.A. in economics and statistics and a first-class honors B.Sc. in electrical engineering, both from the University of Glasgow. He is certified as a Certified Information Systems Security Professional and Certified Information Security Manager.

C. Warren Axelrod, Ph.D.
Delta Risk
P.O. Box 234030
Great Neck, NY 11023
E-mail: waxelrod@delta-risk.net

REFERENCES

1. McGraw, Gary, Software Security: Building Security In, Upper Saddle River, NJ: Addison-Wesley, 2006, pages 213-216.
2. Merkow, Mark and R. Lakshmikanth, Secure and Resilient Software Development, Auerbach Publications, forthcoming September 2010.
3. Anderson, Ross, Security Engineering, Second Edition, Indianapolis, IN: Wiley Publishing, 2008, pages 192 and 212.
4. Hass, Anne Mette Jonassen, Guide to Advanced Software Testing, Boston, MA: Artech House, 2008.
5. Michael, C.C. and Will Radosevich, "Risk-Based and Functional Security Testing," Cigital, Inc., 2005, at <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing/255-BSI.html>
6. Gegick, Michael, "Intro to Security Testing," NC State University, 2006 at <http://resist.isti.cnr.it/free_slides/security/williams/Security1.pdf>
7. "Application-layer security: What it takes to enable the next generation of security services," Nortel White Paper at <http://www.nortel.com/products/01/alteon/2224/collateral/nn105560-010605.pdf>
8. "Application-layer Attack Protection: Proactive defenses for your critical business applications," MacAfee Web Page at <http://www.macafee.com/us/enterprise/solutions/network_protection/application_layer_attack_protections.html>
9. Shane, Scott and Christopher Drew, "Officials Say Iraq Fighters Intercepted Drone Video," The New York Times, December 17, 2009 at <http://www.nytimes.com/2009/12/18/world/middleeast/18drones.html>

NOTES

1. I recall an exceptionally talented QA manager, who reported to me in the 1980s and who would spend a day or so after the functionality of the system had been assured just hitting keys at random to see how the system would respond. He invariably found program errors that had not shown up in the original functional testing. Today this might be called "fuzz testing."
2. See OWASP Top 10 – 2010 (Release Candidate 1) at <http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf>
3. See CWE/SANS Top 25 Most Dangerous Programming Errors at <http://www.sans.org/top25-programming-errors/>
4. There are occasional references to functional security testing such as in Hass's Guide to Advanced Software Testing [4], page 248, Michael and Radosevich's article "Risk-Based and Functional Security Testing" [5], and a presentation by Michael Gegick [6]. In some cases, there is confusion between functional security testing and what the author would term "security functionality testing," which is done to ensure that the security functionality, rather than the application functionality, is correct. Even when functional security testing is defined the same way as it is in this article, as in [4], little detail is provided as to the scope of the testing effort and the use of sampling to make the testing manageable.
5. In the commercial world, "functional testing" is sometimes referred to as "business logic testing."
6. The percentage of attacks affecting the application layer (as opposed to networks) is variously estimated in the 70 to 80 percent range, as in the Nortel White Paper [7] and the MacAfee Website [8]. The accuracy of these numbers is highly questionable since, in the author's opinion, the vast majority of compromises of applications are never detected, especially those promulgated by insiders, such as employees, contractors, and service providers' staff. However other forms of attack are also underestimated. Perhaps the best one can say is that security professionals believe that a large percentage, possibly the majority, of compromises are those related to applications.
7. Descriptions of the various BSI (Build Security In) approaches can be found at <https://buildsecurityin.us-cert.gov/bsi/home.html>, <http://www.bsi-mm.com>, <http://www.opensamm.org>
8. See "Y2K: Overhyped and oversold?" BBC News, January 6, 2000, at <http://news.bbc.co.uk/2/hi/talking_point/586938.stm>
9. In a comment on the author's Bloginfosec column "Insider Threat – Not Knowing That You Don't Know What You Don't Know," available at <http://www.bloginfosec.com/2010/05/10/insider-threat-%e2%80%93-not-knowing-that-you-don%e2%80%99t-know-what-you-don%e2%80%99t-know/2/>, Gary Hinson raises the issue of "plausible deniability." He contends that the "wayward insider" is better able to claim that the unauthorized activity was an accident. This reduces the deterrent value of disciplinary consequences against the employee or other insider.
10. Dr. Herbert (Hugh) Thompson, Chief Security Strategist at People Security, coined this term.
11. Such skills can be demonstrated by testers having appropriate certifications.
12. The author recently spoke with an experienced software engineer, who is involved in the design and development of safety-critical systems. He described the way in which he tests such systems. His approach was very similar to that defined as FST in this article. However, he had not formalized such testing as markedly different from regular functional testing, and he was not aware that this approach was not widely used. This suggests that there are pockets of testers performing FST, but such centers of excellence are not common as demonstrated by the frequency of software failures.

Fault Tolerance With Service Degradations

Dr. Gertrude Levine, Fairleigh Dickinson University

Abstract. The disruptions and/or corruptions that occur during a system's lifecycle require efficient management in order to enable service continuation. We investigate service degradations, which are effective mechanisms for fault tolerance at multiple stages of the anomaly cycle. The acceptance and control of degradations are of particular importance for the prevention of errors.

Introduction

The activation of faults can cause degradations in system services—sometimes tolerable, sometimes intolerable. As long as resulting deviations in system services remain within specified requirements, services can be maintained, although in degraded mode. If deviations exceed acceptable limits, errors occur. As long as erroneous states do not damage component services, error resolution may be possible; at the same time, unaffected states can render service. If errors propagate to component services, component failure occurs; we say that the errors have been activated. Failed components that provide nonessential services can be abandoned. Alternatively, they can be replaced or their corrupted states corrected, assuming sufficient time and resources are available. If component failure prevents the rendering of an essential service, system failure must ensue. Uncontrolled system failure results in faulty products delivered to clients, potentially repeating the cycle of anomalies. We follow the propagation of faults to errors and failures and then to faults again, with service degradation considered as a control mechanism at each stage of the anomaly cycle. Our study applies to both hardware and software systems.

The Anomaly Cycle

A service is a set of outputs and/or inputs together with a set of restrictions (timings, dependencies, and priorities) [3] that satisfy system requirements. Services are rendered to clients for further manipulation and/or for consumption. Precise service requirements may be specified, perhaps for voltage levels, delivery deadlines, or ordering of data, but deviations from optimal specifications frequently are accepted. Delays, truncated services, and fuzzy outputs are all examples of tolerated deviations in some system requirements. We define "service degradations" to be services that are rendered within acceptable deviations from optimal service requirements by system states containing attributes that differ from system specifications for particular conditions under which the service is rendered.

> "ISO 3.5.2 Error: A manifestation of a fault [see 3.5.3] in an object …
> 3.5.3 Fault: A situation that may cause errors to appear in an object.
> A fault is either active or dormant. A fault is active when it produces errors." [1]
> "The adjudged or hypothesized cause of an error is called a fault … A fault is active when it causes an error, otherwise it is dormant." [2]

A fault is a set of attributes that are assigned to system states together with conditional dependency restrictions, yet do not conform to system specifications. A fault is activated when the condition(s) of such a dependency evaluates to true, rendering states unable to provide specified services. If system requirements tolerate deviations from precise specifications, the result can be a degradation of service. For example, consider an unsecured wireless home network. If an unauthorized neighbor eavesdrops, obtains the homeowner's credit card number, and uses it to subsidize a trip to Hawaii, errors have occurred. Suppose, however, that the neighbor's connection only slightly delays the homeowner's service. As long as delays remain within tolerable limits, so that the homeowner continues being serviced, the neighbor has caused a degradation of service. We thus modify the definitions of fault activation that are cited previously: A fault is active when it produces errors or service degradations.

Service degradations are common at multiple stages of a system's lifecycle, not only as direct results of fault activation, but also as by-products of error resolution and component replacement or abandonment. For example, delay degradations occur during error resolution, diversity selection, and fault masking; partial service degradation occurs when nonessential failed components are abandoned; and dependency degradations occur following inferior voting selections of design or data diversity. Service degradations, however, are the only mechanisms applicable for error prevention immediately after a fault has been activated and are relatively efficient because of their ability to be utilized early in the anomaly cycle. Systems monitor deviation patterns to detect suspected degradations, enabling appropriate actions to be taken before

errors occur. Since degradations frequently feed upon themselves, systems must ensure that deviations are limited. For example, channel utilization and packet loss frequency are monitored to forestall errors resulting from network congestion; traffic is monitored in multimedia systems, with the throttling of users, as necessary, to maintain quality of service requirements and prevent errors.

Corrupted (erroneous or failed) service is not service, but disservice. Similarly, intolerable degradations are not degradations, but errors.

> "3.5.2 Error: Part of an object state which is liable to lead to failure."[1]
> "The definition of an error is the part of the total state of the system that may lead to its subse quent service failure. " [2]

An error is a deviation in a service state (caused by fault activation) that renders the state incapable of producing (uncorrupted) service. Service corruption may involve intolerable output values, unauthorized inputs, or unacceptable waits, for example. As long as errors do not corrupt essential services, unaffected states can continue to render service. Some erroneous states are never accessed, i.e., they are implicitly or explicitly abandoned. Others are detected during state changes or state monitoring and resolved before they cause failure. Error resolution is possible only if resulting degradations, such as delays, are tolerable. Errors that are not resolved propagate to those system states that accept their corrupted service. Errors are activated when they cause component failure.

A corrupted state regresses, losing qualities that made it serviceable at that state. (Hardware is frequently serviceable in previous states, such as a demolished building's steel that is reused as scrap or gold jewelry that is melted and reshaped.) The loss of serviceability is critical to the definition of an error, else how do we distinguish between a dormant fault and a dormant error? Both can cause errors and both can lead to "subsequent service failure." Yet, a faulty state can continue to render service; an erroneous state cannot. Consider a system that receives concrete that does not satisfy specifications. The faults in the concrete are not detected during (faulty) acceptance testing. A two-deck bridge is built using the concrete. Under light traffic, the concrete provides optimal service. As the traffic load increases, the concrete bulges, continuing to support traffic but in degraded mode. When stress is applied to the upper deck, the concrete cracks and even light traffic can no longer be sustained. An error has occurred. The lower deck, however, is still serviceable. Then traffic appears on the upper deck. The crack spreads and the entire bridge and its traffic load collapse—a system failure. The upper deck could no longer render service unless the crack was repaired or returned, at the least, to its service state prior to stress application. The provider of the concrete was at fault (and may have incorporated faulty materials that it had accepted). But it was also the responsibility of the clients to properly test the concrete before acceptance. In addition, maintenance crews should have performed necessary repairs, alerted by degradations that became evident during the use of the bridge. We recognize multiple faults, errors, and component failures leading to the failure of the bridge.

> "Failure: The inability of a system or component to perform its required functions within specified performance requirements." [4]

A corrupted state loses its serviceability, but that may not be evident to clients. The acceptance of corrupted service by system states propagates errors; its acceptance by system components causes failure. Components are sets of states that are bound together with dependencies [2] so that they fail and must be abandoned or replaced as a unit. A failed component can be discarded if its service is nonessential. For example, a failed parity disk in RAID 2 systems can be disconnected without loss of input or output service. Alternatively, component failures can be handled by backup and recovery procedures or by component replacement, causing delay degradation. If a component fails, and the service that it provides is essential, and it is neither replaced nor its erroneous states corrected, then the system must fail, i.e., it will deliver corrupted (including missing) service. We say that a system failure is activated when a client accepts its faulty service.

A hazard is an "extraordinary condition" [5] that threatens to destroy all components of a software and/or hardware system. Even systems that have extraordinary defensive mechanisms are vulnerable to some hazards, such as tornados, meteorite landings, or a Linux installation by an inept user. We would not categorize systems as faulty, however, for such vulnerabilities. It is generally impossible or impractical to forestall the execution of each conditional dependency that can render a system inoperable. We claim that hazards cause system failure upon activation, bypassing the states of faults, errors, and component failures. Failure recovery following hazard activation relies on redundancies, where feasible.

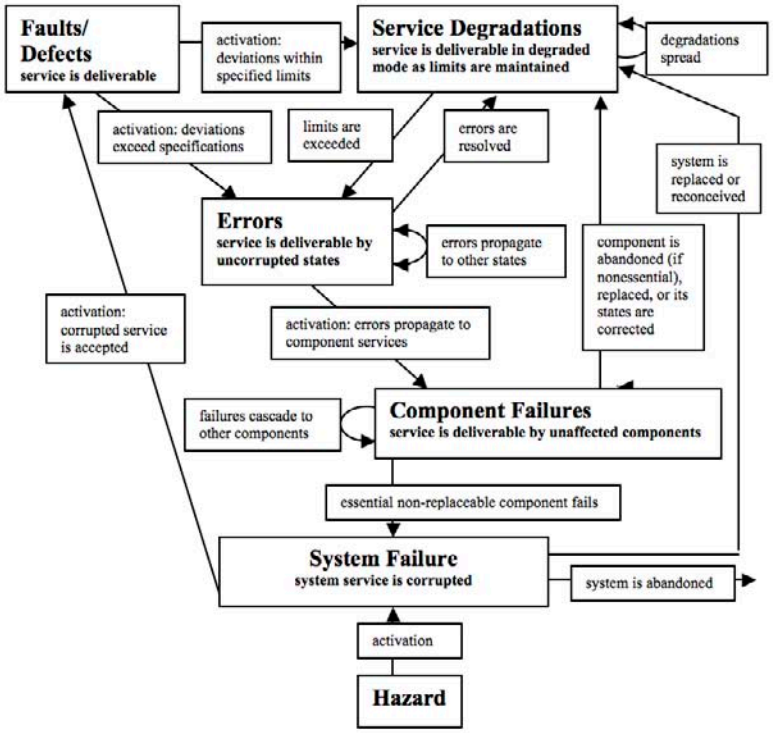
Multiple faults are required for some types of errors (e.g., the errors of security violations [2]); multiple errors are required for some types of component failures (e.g., parity failures following an even number of bit errors); and multiple component failures are required for some types of system failures (e.g., RAID 5 disk failures). Consider the following "fundamental chain" [2] designating the relationship between failures, faults, and errors:

→ failure → fault → error → failure → fault → ...

An expanded diagram of anomaly relationships and propagations should include service degradations and hazards, as well as events that cause transitions between states (see Figure 1 on following page).

Advertent or inadvertent attacks on a system are faulty and exploit (activate) system faults. Some systems adopt onerous procedures in an attempt to control fault activation. These

Figure 1: Anomaly Propagation



constraints are not considered degradations by the system, yet clients may feel differently and cancel the service. Thus, systems seek to minimize the costs of error and failure control, but, since methods are typically heuristics, additional degradations and errors are frequently introduced.

Types of Faults, Service Degradations, and Errors:

A fault, when activated, causes a degradation of service or an error, depending upon whether deviations from optimal service states are within specified requirements. We introduce four classes of faults, errors, and degradations for these anomalies [3], as well as examples of each class:

a. Input/output values: Output and input values can implement data, such as digitally encoded numbers, letters, sounds, images, and odors, or products, such as robotic movements. All output values must be input at a specified location to complete their service, but representations need not always be precise. For example, consider hardware implementations of irrational numbers. These produce deviations from actual values, but usually satisfy client requirements. Perhaps an algorithm is ported to a system that allocates fewer bits for representations; arithmetic overflow can result. Unless exception handling can catch and resolve overflow, perhaps using different numeric representations, output will be erroneous. Or consider defective (faulty) computation that loses precision when summing irrational numbers. If the result remains within specified deviations, output degradation occurs but service can be maintained. The same algorithm might generate an error in an application that requires data of greater precision. As another example, a scratch on an audio disk is a fault. When the disk is played, resulting noise might be considered output degradation. Such noise from a symphony disk is an error that will probably cause the disk to be discarded. Error correcting codes on CDs enable resolution of some noise, but such capabilities are limited. Assume that encrypted data have been input by an eavesdropper via an unsecured wireless connection. If the data are decrypted without authorization and confidentiality is part of system or client requirements, errors have occurred; perhaps the encryption algorithm was faulty or the encryption key was stolen. Still, we claim that the original data states do not lose their serviceability unless output obtained via the decryption process renders them invalid. (A data input with a non-matching key causes a dependency to be assigned to the original data [3]. Unauthorized output of the decoded data conflicts with and renders the original data states unserviceable. Similar mechanisms cause data inconsistency [3] in the lost update problem of databases.) Unauthorized inputs are hardware issues as well, for example, in advertent or inadvertent carbon monoxide poisoning. Output of carbon monoxide into organs causes client failure. Degradations in air quality can signal detectors to assist in failure prevention.

b. Timings: Timing mechanisms can be generalized to count numbers of mappings per interval [3], including metrics such as numbers of allocated resources, transmission rates, and cost overruns. For example, a 56kbps bit rate on a dial-up modem may be considered optimal, while a somewhat lower bit rate is an acceptable deviation. A 56kbps bit rate on a broadband connection is an error, possibly caused by a worm. Firewalls can block worms, but they can cause delays and lost services as they evaluate and block incoming traffic. As another example, consider time and cost overruns, which are common degradations in many development processes. Overruns that exceed specified deviations are errors and have resulted in the cancellation of many projects.

c. Priorities: Priority mechanisms are relevant during competition [3], establishing servicing orders and voting choices. For example, operating systems dispatch high priority processes before competing lower priority processes. If a priority inversion occurs, so that a lower priority process is executed before a dispatchable higher priority process, or before a dispatchable process that blocks a higher priority process, the resulting delay degradation is generally tolerable. If, however, the high priority process has hard real-time requirements, errors and failures will likely ensue. Priority inheritance mechanisms prevent many types of priority inversions. Their implementation in a distributed network, however, can be onerous, causing delay and other degradations. As another example, dynamic network routing algorithms select "shortest" paths using data received from other routers. (They assign priorities based on computed metrics.) Assume that the activation of hardware and/or software faults causes a router failure. Routers executing a faulty routing algorithm may then assign incorrect priorities. If computed paths enable packet delivery within acceptable delays, priority and delay degradation results. If delays are inacceptable and packets are discarded, errors and failures can result. Erroneous routing algorithms may also select paths that do not satisfy system security requirements, potentially causing input errors as well as failures.

d. Dependencies: Interrelationships between system states and components are determined by dependencies. For example, automobiles provide transportation services utilizing

interrelationships between many different components. (Some components, such as video players and coffee cup holders, are nonessential for transportation.) A torn tire may be replaced temporarily with a small spare of lesser quality, causing dependency, as well as output (comfort) and other degradations. If the replacement is also torn, transportation service becomes unavailable. As another example, flexible data structures are implemented with pointers that maintain dependencies between objects. The execution of faulty pointer arithmetic can cause an error in a linked list, so that traversal through a corrupted link must fail. If the list is doubly linked, the traversal algorithm can take the secondary path, resulting in dependency degradation.

All essential components of a system are bound together with a set of dependencies, so that the failure of any component, if not controlled, causes system failure. Dependencies for components of nonessential services are conditional, allowing for their abandonment; then other services can be continued in the degraded dependency mode of partial services [6]. Redundancies enable component replacements to prevent failure. Replacements may be fungible, of lower quality, of higher cost, or even supply alternate services, such as occurs during the degraded dependency mode of emergency services [6]. Replacements are effective using design and data diversity or reflection [7]. Dependency degradation occurs when a replacement component is of lower quality, assuming that the primary component was correctly identified as malfunctioning. Priority degradation, on the other hand, results when a defective voting scheme causes the replacement of a correctly functioning primary component with an inferior product.

Conclusion

Service degradations are the only immediate mechanisms for error prevention after a fault has been activated. The monitoring of degradations and appropriate adjustment of parameters frequently forestalls the occurrence of errors. Systems that augment acceptable deviations in their service requirements, where appropriate, enhance this fault tolerance mechanism.

Degradations of service also occur during error and failure resolution. Recovery is enabled by system requirements that tolerate deviations in acceptable service, such as non-optimal values, non-optimal delivery metrics, non-optimal orderings, or non-optimal service sets. Service degradations are integrated into mechanisms for fault tolerance at all stages of the anomaly lifecycle, with continual efforts to minimize their cost.

Our study of service degradations has yielded a classification scheme and an original diagram illustrating the role of service degradation in the propagation and control of anomalies. We have also introduced amplifications for some commonly accepted definitions. We expect future research to establish a framework for errors and degradations that includes research areas beyond the fields of software and hardware systems.

Acknowledgments

My appreciation to all of the reviewers for their suggestions.

ABOUT THE AUTHOR

Gertrude Levine, Ph.D. Stevens Institute, is a professor of computer science in the School of Computer Sciences and Engineering of Fairleigh Dickinson University. Dr. Levine has been writing a column in Ada Letters called Reusable Software Components since 1990. (One of these columns was published in CrossTalk in March 1992, #32, pp.13-17.) Her research interests include the Ada language and conflict control, specifically in operating systems and networks.

Professor, Computer Science
Fairleigh Dickinson University
1000 River Road
Teaneck, NJ 07666
Phone: (201) 692-2498
Fax: (201) 692-2443
E-mail: levine@fdu.edu

REFERENCES

1. ISO Reference Model for Open Distributed Processing, ISO/IEC 10746-2:1996 (E), 1996, at <http://standards.iso.org/ittf/PubliclyAvailableStandards>.
2. Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. "Basic Concepts and Taxonomy for Dependable and Secure Computing" IEEE Transactions on Dependable and Secure Computing, vol. 1, #1, Jan.-Mar. 2004, 11-33.
3. Levine, G. N. "Defining Defects, Errors, and Service Degradations" ACM SIGSOFT, Software Engineering Notes, vol. 34, #2, March 2009, 1-14.
4. IEEE Computer Society, "Standard Glossary of Software Engineering Terminology" ANSI/IEEE Standard 610.12-1990. IEEE Press, 1990. New York.
5. Goertzel, K. M. "Software Survivability: where Safety and Security Converge" Crosstalk, The Journal of Defense Software Engineering, vol. 22, #6, Sept.-Oct. 2009, 15-19.
6. Mustafiz, S., Kienzle, J., and Berliz, A. "Addressing Degraded Service Outcomes and Exceptional Modes of Operation in Behavioural Models", Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, 2008, pp. 19-28.
7. Rogers, P. "Software Fault Tolerance, Reflection and the Ada Programming Language" Thesis for the Doctor of Philosophy, University of York, October 24, 2003.

An Ecosystem for Continuously Secure Application Software

Mark Merkow, CISSP, CISM, CSSLP, PayPal Inc.
Lakshmikanth Raghavan, CISM, CEH, CRISC, PayPal Inc.

Abstract: A software development ecosystem composed of nine working elements makes it possible to continuously secure application software throughout the entire Software Development Lifecycle (SDLC) and while it's in production use. By orchestrating the activity of these nine elements, organizations and their leadership can reliably and repeatedly produce high-quality software that can stand up to attacks or rapidly recover from intentional or unintentional malicious activity.

Introduction

Provably secure application software can only emerge from a SDLC that treats security as a core element of every phase and in post-deployment. By mandating security within the SDLC itself, management in organizations can rest better at night knowing their infrastructure is continuously working as their defender rather than their enemy. When you address software development as a completed system of phases, tools, activities, and feedback loops, you can bring to life The Rugged Software Manifesto [1] as your deeds match your words.

The U.S. Scheme for software assurance in government and military applications relies on the Common Criteria (CC) Evaluation and Validation Scheme, developed and operated by NIST and the NSA. Critics complain that the CC is too heavyweight and impractical, that it takes too much time, costs too much, and flies in the face of the powerful commercial forces of "time to market" [2].

While CC mechanisms and processes may not be terribly useful for in-house custom developed software applications, many of the concepts and features of the scheme most certainly are. By selectively picking and choosing those software assurance steps from the CC and leading practices in software security, it's possible to build out an infrastructure that produces provably secure application software and provides real-time feedback into the system that forces code with residual vulnerabilities back into the SDLC for rapid remediation and redeployment. A continuously secure ecosystem for software development enables organizations to pay closer attention to building innovative business features and less attention to process or "meta" issues that affect software security and quality.

Catching Errors Sooner Lowers Overall Costs

From the earliest days of software development, studies have shown that the cost of remediating vulnerabilities or flaws in design are far lower when they're caught and fixed during the early requirements/design phases rather than after launching the software into production. Barry Boehm blames late inspection for software errors as the cause of an increase of 40 to 100 times the cost that is required if the errors were caught sooner in the SDLC [3]. Therefore, the earlier you can integrate security processes into the development lifecycle, the cheaper software development becomes over the long haul.

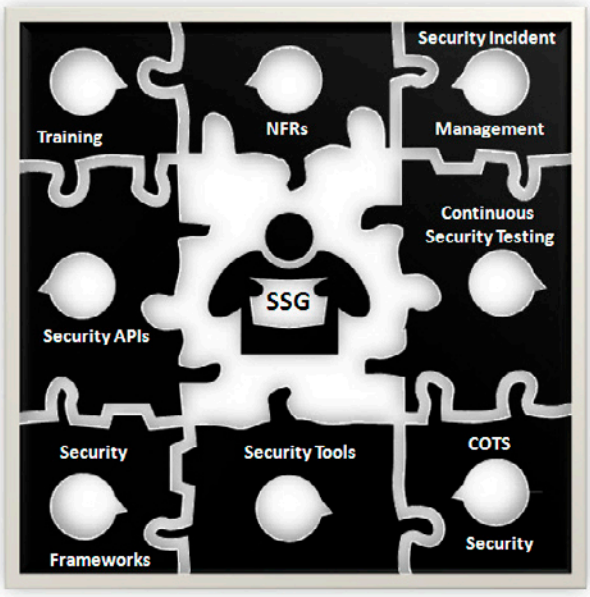
The tricky part is implementing a software development and operational infrastructure in a large enterprise while making it repeatable, scalable, and a natural part of the organization's DNA.

Building Blocks for Continuous Application Software Security

An ecosystem for continuously secure application software requires a robust and reliable infrastructure to make it work. The basic building blocks needed to bring such an infrastructure to life include the following: training and awareness; a Software Security Group (SSG); Nonfunctional Requirements (NFRs); reusable security Application Programming Interfaces (APIs); security frameworks; software security tools; security of COTS software; software security incident management; and continuous security testing.

Once assembled, the complete picture should appear like the one in the diagram below.

Figure 1: Building Blocks for Continuously Secure Application Software



The SSG plays a central role in the ecosystem, providing a crucial source for strategies, implementation, orchestration, and governance for the tools and processes needed to continuously improve the overall security of the software being developed [4].

1) Training and Awareness

While training may not fit directly into any particular SDLC phase, it plays a crucial role in improving the overall security and quality of software. Training is a prerequisite for anyone who has any role anywhere in the software development environment.

All developers and other technical members of the software design/development/test teams should undergo security training that explains the responsibilities of their role, establishes the expectations for their part for security, and provides best practices and guidance for developing and maintaining high-quality software.

2) Software Security Group

A formal SSG, having its primary responsibility be the improvement of security of the SDLC and the software it produces, is a core element. The role of the SSG includes the following: define the software security strategy; develop the processes for integrating security into all the phases of SDLC; roll out software security tools for developers and testers; establish security testing processes; oversee the development of security APIs and frameworks; develop and deliver software security awareness/training sessions; and define, track, and report the metrics related to the success and progress of the overall software security program.

3) NFRs

What software is expected to "do" is described by users in functional requirements. These requirements show up in the early development phases when a group of would-be users collect to describe what they want. NFRs are the quality, security, and resiliency aspects of software that only show up in requirements documents when they're deliberately added. These requirements are the outcome of software stakeholders who meet to discuss the planned software. These stakeholders should include the people who will use the system, the people who will operate it, the people who will maintain it, the people who will oversee the governance of the software development lifecycle, security professionals, and the legal and regulatory compliance groups who have a stake in assuring that the software is in compliance with local, state, and federal laws.

The key to a successful software security program is to establish a requirements analysis process within the SDLC that treats nonfunctional requirements as equal citizens to functional requirements. The SSG should establish processes to assure that this regularly occurs and help the application development organization to put together well defined and reusable NFRs. For example, if your organization follows the agile development methodology, "user stories" [5] are one method for requirements collection that you can quickly apply. Stakeholders could use these methods to capture nonfunctional requirements as well as functional requirements. Some of the key NFRs that must be considered include availability, capacity, efficiency, extensibility, interoperability, manageability, maintainability, performance, portability, privacy, recoverability, reliability, scalability, security, and serviceability.

4) Reusable Security APIs

Application developers have no business writing security functions. Using security controls is different from building them. A better bet is to build and promulgate standardized security APIs for developers to re-use and integrate into their applications. These APIs perform the most important security functions such as validation, encoding/decoding, cryptographic processes like encryption, hashing, authentication, authorization, logging, error handling, etc. The Open Web Application Security Project (OWASP) Enterprise Security API [6] is one such API that any organization can adopt and customize for its software development and operational processes. Developers will need education and training on using these security APIs and should be prevented from developing their own.

5) Security Frameworks

In addition to reusing security APIs in custom development work, security frameworks can help to automatically prevent many well-known attacks like Cross-site Scripting [7], Cross-site Request Forgery (CSRF) [8], and others. These frameworks are built by a centralized application security development under the guidance of the SSG and deployed to all production Web applications. These frameworks automatically provide security functions that counter well-known Web attacks. Many existing frameworks like Spring, Struts, etc., have some security built-in, but controls are frequently missing, incomplete, or wrong. After analyzing the gaps in each framework, some of the typical custom frameworks that can be built and deployed in any application infrastructure include: output encoding framework; input validation framework; and CSRF framework.

Whereas APIs need to be explicitly used by developers, these frameworks work invisibly and require no explicit action from the developers. If developers fail to output encode the parameters sent to an HTML page, these frameworks will automatically do it for them.

6) Software Security Tools

Static Analysis of source code and Dynamic Analysis of runtime modules provide significant value to any development environment that honors secure applications. Developers need regular access to static analysis tools for source code analysis and to report security vulnerabilities. Quality Assurance (QA) (testing) teams also require access to dynamic analysis tools (also called black-box testing tools) for complete code coverage. While we don't recommend any specific vendors, we do recommend that organizations perform an objective evaluation of all available tools and select the ones that would work well with their own environment and processes. Some existing reviews done by NIST SAMATE Project [9] and NAVSEA [10] can be a good starting point to begin the evaluation.

While there are multiple strategies to scan source code using any of the commercially available scanners (e.g., Ounce, Fortify, etc.), we strongly recommend a two-pronged approach to the deployment model: an Integrated Development Environment (IDE)-based version for developers to use at their desktops, and a build process integration for effective governance and management of the SDLC. With an integrated scan that runs automatically when an application is submitted for a QA Environment Build Operation, management can define gating criteria that routes the application to the appropriate channels based on the outcome of the source code scan.

IDE Integration for Developers:

To help developers scan the code they write early enough in the lifecycle, you need to provide them with unfettered access to automated scanning tool(s) so that they can perform scans themselves, via an IDE running at their desktop computer.

Scanning can be performed on a single function or method, on a file or a collection of source code files, or on the entire application system. This self-service scan will provide results that developers can use directly to clean up their code based on the findings. The scan report typically provides generic recommendations on how to fix the identified vulnerabilities as well. The OWASP Web Testing Environment (WTE) Project [11] is one example of testing tools for developers that aims to make application security tools and documentation readily available. The WTE has several other goals too, including to: provide a showcase for popular OWASP tools and documentation; provide the best, freely distributable application security tools in an easy-to-use package; ensure that the tools provided are as easy to use as possible; continue to add documentation and tools to the OWASP WTE; continue to document how to use the tools and how the tool modules were created; and align the tools provided with the OWASP Testing Guide.

Build Integration for Governance:

Build process-based scanning occurs when the entire application (all modules and libraries) are ready to be built and prepared for QA testing. This typically includes source code components originating from different development teams and/or different software development companies (e.g., outsourced development shops). This centralized scanning is meant as a governance and management mechanism and can be used to provide gating criteria before the code is released to the next phase in the SDLC. Typical gating criteria for production movement might include zero high-risk vulnerabilities; no more than five medium-risk vulnerabilities; no more than 10 low-risk vulnerabilities, etc.

A software supply-chain risk can be a defect in the delivered software or in the default installation or configuration that an attacker can exploit [12] and a build-governance software scan can help to uncover and eliminate errors that otherwise would fall through the cracks.

You should use the build process-based scanning not only for planned software releases but also for emergency bug fixes. Since the scanning process is closely integrated with the build process, automation takes care of assuring that source code scanning happens every time. When the assurance level of the automated scanner is high (not too many false positives), then the build server can be triggered to fail the build based on the gating criteria and send the application back for remediation.

Metrics that are useful to track for measuring performance and progress could include: number and percent of applications scanned using IDE scan; number and percent of applications scanned using Build scan; number and percent of applications scanned using Build scan that failed/passed; vulnerability density (vulnerabilities/thousand lines of code); vulnerability severity comparison across projects or development teams; vulnerability category comparison across projects or development teams; vulnerability category-specific trending; average time taken to close high/medium/low-risk vulnerabilities; vulnerability distribution by project; and top 10 vulnerabilities by severity and frequency.

Dynamic Analysis (Black-Box Testing) During QA Testing:

In addition to functional testing of an application in the QA environment, security testing using a black-box scanning tool (like IBM's AppScan) can help to catch any remaining vulnerabilities that fell through all prior safety nets.

A bug priority matrix for the organization should include the definitions of security defects that enable the QA team to create separate security defect records and help classify their priority. The testing carried out by this independent team might also serve as gating criteria for promoting the application from QA testing to the production environment. The results from these test results should be shared with the developers soon after the tests are run, so the developers can develop strategies for remediating the issues that are uncovered. Once the criteria for moving an application to production are met, the QA team should sign off on the security vulnerability testing, along with the other test results. Black-box testing also ensures that other minor feature additions and bug fixes are also tested for security bugs before they too are moved to production. Furthermore, centralized testing yields meaningful metrics as experience with the tools is gained and progress (or regress) can be measured and reported over time.

7) Security of COTS Software

When COTS software is used by custom-developed systems or offered as an infrastructure service, you may run into problems when you discover vulnerabilities during preproduction black-box testing and penetration testing. In most cases, when problems are found with COTS systems, it's difficult to identify what to do about them or even determine who to contact. As users of COTS products, information protection and risk management professionals are too far removed from the product development activities. Today's state of COTS security testing often leaves software buyers with little ability to gain the confidence they need to deploy business-critical software. In its absence we are forced to develop our own countermeasures and compensating controls to counter these unknown potential threats and undocumented features of the software. As we mentioned in the beginning of the article, because of any actual or perceived shortfalls of the CC, commercial businesses are forced into using various other and related approaches to gaining confidence in the security of COTS products. Here we take a look at two common commercial approaches.

ICSA Labs:

The ICSA Labs certification is based on public, objective criteria that yield a pass/fail result [13]. The criteria–drawn from expertise across the industry–are clearly defined and address common threats and vulnerabilities for each COTS product. The criteria are applicable among products of like kind and can therefore be used for better understanding, comparing, and assessing security products.

Veracode's VerAfied Software Assurance:

Delivered as a cloud-based service, Veracode's VerAfied process yields an analysis of final, integrated applications (binary analysis) and provides a simple way to implement security best practices and independently verify compliance with internal or regulatory standards without requiring any hardware or software [14].

8) Software Security Incident Management

The SSG must work closely with other stakeholders like the Security Operations/Monitoring team, Application Development teams, etc. to put together a well-defined application security incident management process. Even with controls throughout prior phases of the SDLC, bad code still manages to wind up in the production environment. A Security Incident Management Process is needed to analyze, triage, and apply short-term restoration fixes (such as application firewall rule changes), coordinate long-term code level changes, and validate and deploy security fixes.

In his September/October 2010 CrossTalk article, "The Balance of Secure Development and Secure Operations in the Software Security Equation," Sean Barnum advocates a holistic approach that balances secure software development and secure IT operations [15]. By including the appropriate participation from the development community, the incident management process acts as the final safety net to protect the organization from further damage once an incident is declared.

Some key success factors include the following: appropriate stakeholder access to reports from continuous security testing and monitoring tools; a unified bug tracking system that everyone uses and provides end-to-end tracking and closure of identified security bugs; and well-defined processes to identify appropriate source code owners to alert and engage them about production vulnerabilities and to help develop, test, and deploy security fixes.

9) Continuous Security Testing

The last piece of the puzzle completes the picture and pulls together all the elements that compose the ecosystem. Continuous testing using regularly updated black-box scanners set to run automatically can help to assure that new vulnerabilities, and those missed in prior development phases, are caught and acted upon before anyone on the outside has the opportunity to find them and exploit them. Scans can be scheduled based on any number of factors related to the application.

Continuous security testing in the QA Environment of production-released code, along with a well-defined feedback loop that relies on the software security incident management process, can alert the application owners about residual security problems so they can be addressed immediately. As security incidents are opened, the application is forced back into the analysis or design phase of the SDLC and works its way back through the SDLC, helping to assure that software is never released and forgotten.

Conclusion

With a completed puzzle of symbiotic and synergistic elements working in concert, you can implement a well-orchestrated, well-oiled feedback system that over time will improve the SDLC itself as experience is gained and processes and tools are fine-tuned. Meeting the pledge of The Rugged Software Manifesto includes improving the software development environment itself as you improve your own skills. By inculcating security activities and features into the entire SDLC and beyond, you can rest assured that you're doing all you can to address and reverse the scourge of insecure application software.

ABOUT THE AUTHORS

Mark Merkow, CISSP, CISM, CSSLP works at PayPal Inc. (an eBay company) in Scottsdale, AZ as a manager in the IT Security Department. He has over 35 years of experience in Information Technology from a variety of roles, including Applications Development, Systems Analysis and Design, Security Engineer, and Security Manager. Mr. Merkow holds a master's in Decision and Information Systems from ASU, a master's of Education in Distance Learning from ASU, and a bachelor's degree in Computer Information Systems from ASU. He chairs the Financial Services Information Sharing and Analysis Center Education Committee, serves on the BITS Security Working Group and the Research and Development Committee of the Financial Services Sector Coordinating Council on Homeland Security and Critical Infrastructure Protection. Mr. Merkow has authored or co-authored 10 books, including his latest, "Secure and Resilient Software Development" (2010, Auerbach Publications).

Mark Merkow
9999 North 90th Street
Scottsdale AZ 85258
E-mail: mmerkow@paypal.com
Phone: (480) 862-7391

Lakshmikanth Raghavan
2211 North 1st Street
San Jose CA 95131
Email: lraghavan@paypal.com
Phone: (408) 967-4637

Lakshmikanth Raghavan (Laksh) works at PayPal Inc. (an eBay company) in San Jose, CA as Staff Information Security Engineer in the Information Risk Management area. He has over nine years of experience in the areas of information security and information risk management and has been providing consulting services to financial services companies around the world in his previous engagements. He is a Certified Ethical Hacker and holds the CISM and CRISC certifications from the Information Systems Audit and Control Association. Laksh is the co-author of "Secure and Resilient Software Development" and holds a bachelor's degree in Electronics & Telecommunication Engineering from the University of Madras, India.

REFERENCES

1. Rugged Software. Web. 30 Sept. 2010. <http://www.ruggedsoftware.org>.
2. Merkow, Mark S., and Lakshmikanth Raghavan. Secure and Resilient Software Development. Boca Raton, FL: CRC, 2010. Print.
3. Boehm, Barry W., and Richard Turner. Balancing Agility and Discipline: a Guide for the Perplexed. Boston:MA, Addison-Wesley, 2006. Print.
4. The Building Security In Maturity Model (BSIMM). Web. 30 Sept. 2010. <http://bsimm2.com/index.php>.
5. Introduction to User Stories <http://www.agilemodeling.com/artifacts/userStory.htm>
6. "Category: Enterprise Security API." ESAPI. OWASP. Web. 30 Sept. 2010. <http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API>.
7. "Cross-site Scripting." Wikipedia, the Free Encyclopedia. Web. 30 Sept. 2010. <http://en.wikipedia.org/wiki/Cross-site_scripting>.
8. "Cross-site Request Forgery." Wikipedia, the Free Encyclopedia. Web. 30 Sept. 2010. <http://en.wikipedia.org/wiki/Cross-site_request_forgery>.
9. NIST SAMATE - Static Analysis Tool Exposition. Web. 11 Nov 2010. <http://samate.nist.gov/Main_Page.html> & <http://samate.nist.gov/SATE.html>
10. Software Security Assessment Tools Review. Web. 11 Nov 2010. <https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Tools-Paper-2009-03-02.pdf>.
11. OWASP Web Testing Environment (WTE). Web. 15 Nov 2010. <http://code.google.com/p/owasp-wte>.
12. Ellison, Bob. Supply-Chain Risk Management: Incorporating Security into Software Development. 03 2010. DHS National Cyber Security Division. 15 Nov. 2010 <https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/acquisition/1140-BSI.html>.
13. ICSA Labs. ICSA Labs. Web. 30 Sept. 2010. <https://www.icsalabs.com>.
14. Application Security | Veracode. Veracode. Web. 30 Sept. 2010. <http://www.veracode.com>.
15. Barnum, Sean. The Balance of Secure Development and Secure Operations in the Software Security Equation. Crosstalk – The Journal of Defense Software Engineering. Vol 23 Number 5. Sept/Octorber 2010.

Ensuring Software Assurance Process Maturity

Edmund Wotring III, Information Security Solutions, LLC
Sammy Migues, Cigital, Inc.

Abstract. All organizations–government and commercial alike–share an interest in minimizing their software vulnerabilities and, consequently, in maturing their software assurance capabilities. Successful software assurance initiatives require organizations to perform risk management activities throughout the software lifecycle. These activities help to ensure organizations can meet software assurance goals, including those related to reliability, resilience, security, and compliance. The Software Assurance (SwA) Checklist for Software Supply Chain Risk Management (hereafter referred to as the SwA Checklist) serves as a framework to help organizations establish a baseline of their risk management practices and select maturity model components to better meet evolving assurance goals.

Introduction

Software assurance is the level of confidence that software is free from vulnerabilities, whether intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that it functions in the intended manner.[1] Once an organization becomes aware of the need to meet software assurance goals, the next step is to assess its current development and procurement activities and practices. Such an analysis requires at least two things. The first is a repeatable and objective assessment process. The second is a clear benchmark or target that represents a suitable level of risk management given the nature of the organization and the software's mission. Performing this assessment periodically provides an ongoing understanding of the maturity of respective software assurance capabilities.

Choosing a methodology for appraising an organization's ability to meet software assurance goals may seem overwhelming because there are several maturity models available, each with their own focus and level of granularity. For an organization that may be new to the area of software assurance, it can be a challenge to simply find good sources of guidance, much less understand which parts of each model are best suited for its environment and supply chain. Although finding the right maturity model may seem challenging, organizations should not wait for an authority to mandate a software assurance initiative. Such mandates are typically intended to be "one-size-fits-all" and offer limited flexibility. Organizations are best served by tailoring a software assurance strategy to their own supply chains.

Selecting the best maturity model, or model components, for a particular organization to begin addressing assurance goals may also present a time-consuming learning curve. In order to facilitate an understanding of how multiple maturity models address similar assurance goals, the authors created a model-agnostic framework as part of participation in the SwA Forum Processes and Practices (P&P) Working Group (WG), which is co-sponsored by organizations with DHS, DoD, and the National Institute for Standards and Technology. This analysis involved mapping maturity models, and their respective practices, within the framework. The agreement among the models provides a valuable reference. This framework evolved into the SwA Checklist, which serves as a model-agnostic harmonized view of software assurance guidance.

The SwA Checklist can help organizations begin a dialogue amongst the entities in the supply chain that influence and/or support the software throughout the lifecycle. Using the checklist to characterize each of the organizations in a given supply chain provides extraordinary insight into the credibility or trust deserved by a given piece of software. By leveraging this insight, organizations can verify implicit assumptions that certain practices are taking place and align their activities with assurance goals to mitigate risks within their supply chains. Organizations can also use the checklist to organize evidence for assurance claims while assessing all of its practices as it performs the activities necessary to complete its baseline. Finally, organizations can use the baseline to engage their senior leadership regarding the areas in which resources are needed to meet assurance goals based upon guidance from the mapped models.

The SwA Checklist provides a consolidated view of current software assurance best practices in the context of an organized SwA initiative. The checklist is currently implemented as a "hot linked" Microsoft Excel spreadsheet that provides a cross-reference of goals and practices with side-by-side mappings to several publicly available maturity models. Organizations can use the mappings to identify where the maturity models agree and diverge, and use this consolidated format to select model components best suited to their environments.

Once an organization establishes its assurance goals, selects a maturity model (or model components), and captures its baseline, it can then establish an improvement plan for achieving software assurance goals as it develops and/or acquires secure software. Working with its direct customers (downstream in the supply chain) and suppliers (upstream in the supply chain) to improve software assurance will have a large multiplier effect as the approach spreads to other organizations.

Intended Use

The intended users of the SwA Checklist are organizations that currently are or soon will be acquiring or developing software. Organizations may have many options when developing or acquiring software from various sources. Although vendors and developers may offer software that meets specified functional requirements and provides myriad features, these offers are inconsequential if the data and functions are not protected. Developers and acquirers must give significant consideration to the ability of the software to reliably function and protect data and processes over the life of the product. Organizations can use the SwA Checklist to guide their own development or to evaluate vendor capabilities. Organizations can use the baselines they establish to facilitate an understanding of similar assurance goals and practices among several freely available maturity models, which can help guide the selection of the most appropriate model components.

Design of the SwA Checklist

The SwA Checklist is available at no cost at <https://buildsecurityin.us-cert.gov/swa/proself_assm.html>. The SwA Checklist is currently being vetted and we request your feedback based upon practical use within the field. A feedback form is available at the same URL above. The authors designed the checklist to be understandable by users with various levels of SwA experience (readers are invited to download a copy now and review it while reading this section).

The SwA Checklist contains multiple tabs/worksheets including the following: Intro, SwA Checklist, Sources, BSIMM, CMMI-ACQ, OSAMM, PRM, and RMM. The "Intro" tab serves as the introductory section that also provides pointers to each of the included models. The "SwA Checklist" tab provides the information that enables users to perform their analysis. Content from the included models is organized into five domains: Governance, Knowledge, Verification, Deployment, and Supplier Management. This categorization helps to harmonize terminology and makes it easy for the user to locate specific guidance. Within each domain are three categories containing a short, high-level goal and a set of three corresponding practices. There is a "Status" cell under each practice. Users can click on the cell to open a pull-down menu with pre-defined responses to input their organization's implementation status for each practice. The range of possible status levels in the pull-down menus includes the following:

- **Unknown**
- **Not Applicable**
- **Not Started**
- **Partially Implemented Internally**
- **Partially Implemented by Supplier(s)**
- **Partially Implemented Internally and by Supplier(s)**
- **Fully Implemented Internally**
- **Fully Implemented by Supplier(s)**
- **Fully Implemented Internally and by Supplier(s)**

It is the combination of the status of each practice that will help an organization understand its ability to execute on software assurance activities in development and acquisition.

SwA Tools Relationship

Another tool that is mapped to multiple maturity models, the SwA Self-Assessment, is also available on the same webpage on the DHS SwA Community Resources and Information Clearinghouse website. The SwA Checklist and the SwA Self-Assessment are resources made available from the SwA Forum. The tools provide alternative views on similar assurance process frameworks whose shared objective is software improvement. It is in an organization's best interest to try both approaches and use the one that works best for its own environment. No matter which tool users select, it is important to remember the ultimate goal is producing and delivering rugged software.

The implementation status options vary based upon the degree to which the practice is implemented (i.e., not started, partially implemented, or fully implemented) and the party responsible for each practice (i.e., internally, by the supplier, or by both). The two other responses included in the pull-down menu are "Unknown" and "Not Applicable." The user should follow up on any response marked with either of these statuses. Organizations should mark a practice "Unknown" if it is unknown whether someone is performing the practice or who is responsible for performing it. Such a practice is almost certainly an area of increased risk and requires further investigation. Likewise, if a practice is marked as "Not Applicable," the user should obtain justification for selection of that status. Supply chain partners must understand the environment in which the software will be deployed and meet the end customers' assurance needs even if those needs are not explicitly stated. When assurance goals are analyzed from such derived requirements, certain practices may reveal themselves as applicable. Thoroughly investigating the status of each practice is a valuable due diligence exercise that may result in the user discovering that certain practices actually are applicable or that practices are already being performed as part of other related practices.

By performing the analysis required to assign a status to each practice, the user gains a greater understanding of their overall supply chain and establishes an assurance baseline. This understanding will enable more productive dialogue among all supply chain parties and will foster better understanding of where risk is introduced during acquisition or development of software.

Maturity Model Mappings

The third tab of the spreadsheet, Sources, includes all the same goals and practices from the SwA Checklist tab. Table 1 contains a portion of this view. The Sources tab also includes mappings for each practice to several maturity models, described in the sidebar to this paper on page No. 32 titled Maturity Models (Maturity Models Mapped within the SwA Checklist). All mappings are hyperlinked to other tabs in the spreadsheet. Clicking on a hyperlinked mapping will take the user to the related section on the tab for the corresponding maturity model. The user can return to the Sources tab by clicking on the hyperlinks in column A of any of the maturity model tabs.

There are several benefits to viewing the mappings for each practice in the SwA Checklist side-by-side in the Sources tab. The mappings help the user to see how the maturity models agree and diverge on each of the related practices. Since each model has its own particular focus, viewing the relationships

Maturity Models

There are several freely available maturity models that focus on securing software. Each has its own focus and level of granularity. The publicly available maturity models mapped in the Sources tab of the SwA Checklist include:

- Building Security In Maturity Model version 2 <**http://www.bsimm.com**>
- Carnegie Mellon University SEI CMMI® for Acquisitions, version 1.2 <**http://www.sei.cmu.edu/cmmi/index.cfm**>
- Open Web Application Security Project Open Software Assurance Maturity Model version 1.0 <**http://www.opensamm.org**>
- Software Assurance Forum Processes and Practices Working Group Assurance Process Reference Model, September 2010 <**https://buildsecurityin.uscert.gov/swa/downloads/20100922_PRM_Practice_List.pdf**>
- Carnegie Mellon University/CERT Resiliency Management Model, version 1.0 <**http://www.cert.org/resilience/rmm.html**>

The authors performed a model-agnostic analysis to determine how these maturity models help organizations address assurance goals and practices and to determine where the models converge and diverge. This analysis of the mappings between the models revealed a high degree of agreement. Organizations can use the checklist to determine process improvement opportunities and establish a baseline from which to benchmark their capabilities. More information on the maturity models analyzed and included in the SwA Checklist is available at <https://buildsecurityin.us-cert.gov/swa/proself_assm.html>.

among them provides a context from which the user can better understand the assurance goals and practices. The user will also see how various models address similar goals and practices. This will help the user begin selecting a maturity model that will be of most use to their particular software assurance needs.

Table 1: Sources Tab Snapshot

	Governance		
	Strategy & Metrics	Policy & Compliance	Training & Guidance
Practices	Establishes Security Plan; communicates and provides training for the plan	Identifies and monitors relevant compliance drivers	Conducts security awareness training regularly
BSIMM	SM1.1	CP1.1	T1.1
	-	CP1.2	T3.4
CMMI-ACQ	PP SG2 – SG3	OPF SG1	OT SG2
	-	-	-
OSAMM	SM1B	PC1A	EG1A
	-	PC1B	-
PRM	SG 2.1	SG 3.1	SG 1.3
	SG 1.3	-	-
RMM	RTSE: SG2 – SG3	COMP: SG2	OTA: SG1 – SG2
	MON: SG1	MON: SG1 – SG2	-

Appraisal Considerations

When performing an appraisal using the SwA Checklist, it is important that the user adapt the checklist to the processes being performed and the structure of their organization's supply chain. Users may determine that they implement a different practice that also supports an assurance goal in the checklist. This is typical since not all organizations employ the same practices despite desiring roughly the same assurance goals. Users may also perform an evaluation of a supplier or a division of an organization that only manages a portion of the processes in the overall supply chain. In this case, it is likely that not all the goals and practices within the checklist will apply to this specific supplier or division. Users should leverage the SwA Checklist to determine whether they are taking a comprehensive approach to produce rugged software throughout the entire supply chain. This approach may require evaluating multiple suppliers, divisions, and other entities to comprehensively manage risks and to ensure supply chain partners meet assurance goals.

The mappings of the models in the Sources tab provide valuable reference and context as users complete a baseline. As users become more aware of how the models address similar goals and practices, they may begin to find currently unimplemented model components that are useful for their environments and specific assurance needs. The models referenced within the checklist are designed with varying levels of granularity ranging from high-level control objectives to lower level controls. Each of these perspectives may provide insight into addressing the assurance challenges in various supply chain environments.

Baseline Summary

After users establish a baseline, a summary displays at the bottom of the SwA Checklist tab. This summary depicts a count of each category of implementation status and is highlighted in a conditional formatting color scheme according to the following:

"Not Applicable" practices – Grey
"Unknown" and "Not Started" practices – Red
"Partially Implemented" practices – Yellow
"Fully Implemented" practices – Green

This system provides an easy-to-view dashboard for an organization's overall implementation of practices.

The color-coded system provides a way to quickly assimilate data contained within the user-created baseline. Although the system uses stoplight colors, improvement efforts should not focus solely on the "reds" and "yellows." A practice highlighted in green does not necessarily satisfy the organization's assurance goals or adequately mitigate risks. Further, a practice highlighted in green is one that is being performed, not necessarily one that is required. Organizations must analyze the entire checklist to determine if the correct entity performs each practice correctly and to a sufficient extent, and if each practice is actually mitigating risks according to the organization's assurance goals. Only after determining these factors can the organization outline a plan to effectively and efficiently improve its software assurance capabilities.

Common Appraisal Challenges

The most common issue users face when creating a baseline pertains to practices for which the status is "Unknown." In these instances, the best approach may be to document the process flow surrounding the practice. It is helpful to coordinate with the parties involved in processes surrounding the practice to determine the degree to which the process is implemented. Determining responsibility for each practice is another common issue faced by users. Appraisers should diligently clarify accountability and responsibility during their analyses. The third frequently arising issue is tracking execution of software assurance activities and ensuring suppliers and acquirers do them consistently and effectively. Even when users know what practices are implemented and who is responsible for them, they may be unaware how well they are implemented. Lastly, if users know a practice is implemented, who is responsible for its implementation, and whether it is executed correctly, they still may not know whether it is effectively reducing risk and should be continued.

Even though the practices marked as "Fully Implemented" on the checklist will register as green, this does not necessarily mean they represent money (or resources) well spent. It is important for organizations to select components from the source models to improve the implementation of practices specifically required to meet assurance goals and to ensure their satisfactory completion. It is important to measure not only the assurance activities, but also the software lifecycle artifacts (e.g., code) to ensure both are improving. Overall, organizations should determine the model components that help them accomplish a coherent and cohesive set of activities that accomplish organizational goals based upon business objectives and risk appetite.

Conclusion

Establishing an implementation baseline of the practices within an organization's supply chain will foster a better understanding of its true capability to develop, acquire, and deploy secure software. Using the checklist, an organization may identify opportunities for improvement and begin to create a plan to address improvement areas by selecting model components from the mapped maturity models. The more robust the processes are surrounding software lifecycle processes, the more likely an organization will develop and acquire truly rugged software. The SwA Forum P&P WG plans to periodically update the SwA Checklist to ensure it aligns with updated versions of the models mapped in the Sources tab and to incorporate other models into this mapping in the future.

Acknowledgements

This work is funded in part by the DHS Software Assurance Program. Many colleagues and members of the Software Assurance community provided valuable feedback on the checklist and this article including: Joe Jarzombek, DHS; Don Davidson, OASD-NII / DoD CIO; Michele Moss, Booz Allen Hamilton; Lisa R. Young, CERT; Walter Houser, SRA; Doug Wilson, Mandiant; Rama Moorthy, Hatha Systems; and Dr. Robin A. Gandhi, Nebraska University Center for Information Assurance.

ABOUT THE AUTHORS

Edmund Wotring III is a Senior Security Engineer with Information Security Solutions, LLC. He previously supported various federal government clients with security compliance and process improvement initiatives. He has advised senior leadership to ensure compliance processes facilitate effective security. He currently supports the Department of Homeland Security National Cyber Security Division's Software Assurance program.

E-mail: ed.wotring@informationsecuritysolutionsllc.com

Sammy Migues is a Principal and Director of Knowledge Management at Cigital. He has nearly 30 years experience performing security research and providing practical solutions to government and commercial customers. He is currently working on expanding the BSIMM research, smart grid security demonstration projects, new methods of software security training, and helping organizations start or grow software security initiatives.

E-mail: smigues@cigital.com

NOTES

1. Committee on National Security Systems (26 April 2010). CNSS Instruction No. 4009. National Information Assurance (IA) Glossary. [Accessed 02 Nov 2010]. Available from: <http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf>.

Disclaimer:

® CMMI is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

What Is Your Least Favorite Security Question?

Password saturation is a bane of the 21st century. Passwords are required to access pay statements, approve leave, buy music, change insurance deductibles, bid on auctions, check social networking sites, check e-mail, and so on.

To alleviate the inconvenience of inevitably-forgotten passwords, some sites use security questions to differentiate between legitimate users and hackers trying to break into that user's account. This is a sound idea; however, its implementation is often fundamentally flawed. There are too many bad security questions, and far too few good ones. System architects have simply not done a good job designing foolproof questions.

Take my (least) favorite security question, "What is your favorite color?" At first glance, this may seem sensible. But I don't have one favorite color; my favorite colors vary with the seasons. In October, I love the warm glow of a burnt orange; in December, a Spruce green; in springtime, the brilliant magenta-pink on my redbud tree. Besides, when I'm asked to name a favorite color, I typically wonder, "Am I buying a new necktie, new underwear, a new sofa, or a new car?"

Questions about "favorites" make rather crummy security questions because "favorites" fluctuate according to mood, situation, and circumstances.

Questions that prod me to remember my childhood are particularly vexing too. Not that my childhood was traumatic (it wasn't), but my memory is getting faulty. Ask about my "first" or "favorite" anything from childhood, and I'm likely to give you a different answer eight months later. "What was your favorite Christmas present as a child?" What kind of question is that?

Consider the list of available security questions at the Air Force's Advanced Distributed Learning Service. Users of that system must select not two, not three, but six security questions, from a list that references cartoon characters, friends, sports, TV shows, and food. Here's the complete list, along with my initial reactions (which invoked great angst, upon realizing I'd be lucky to remember even *one* of these answers):

"Who was your favorite cartoon character as a child?" (Do *The Simpsons* characters count? Or only Itchy and Scratchy?)

"What was the name of your best friend as a child?" (I had a new best friend every time I changed schools.)

"What was the name of your favorite teacher in high school?" (Do I still have to call her Miss Simmons? Or can I call her Ann now?)

"What was your first job?"
(Which came first: babysitting or cutting grass?)

"What was your favorite sport as a teenager?"
(To watch, or to play?)

"What was your favorite TV show as a child?" (All that time wasted in front of the TV and I have to pick just one?)

"What was your first pet's name?" (Do goldfish and hamsters count? What about pet rocks?)

"Who was the first presidential candidate for whom you voted?" (In the primaries? Or in November?)

"What was the make and model of your first car?"
(First car I drove? Or first car I owned?)

"Who was your childhood hero?" (Fictional? Or role model?)

"Who did you go with or take to your high school prom?"
(Which prom? I went twice.)

"If you had chosen your first name, what would it have been?"
(When people choose their own names, we get names like Lady Gaga.)

"Who was your first love?" (Do I still have to call her Miss Simmons? Or can I call her Ann now?)

"What is your favorite American landmark?"
(Can Daytona Beach count as a landmark?)

"What is your favorite Science Fiction character?"
(I don't remember the names of Asimov's robots.)

"What is the name of the first National Park you visited?"
(I remember getting carsick, but I don't remember the name of the park.)

"What was your favorite food as a child?"
(Breakfast, dinner, or dessert?)

"What is the name of your favorite sports team?"
(The last home team to win a championship.)

As a software engineer, I'm amazed when such inane questions get past design reviews and acceptance testing to the point where literally thousands of government workers must scratch their heads in frustration, spending countless man-hours deliberating about which program they liked more as a child (was it *Mork and Mindy*? or *Star Trek*?), then spend even more time debating coworkers about whether or not Wii counts as a sport.

Personally, I prefer questions that fluctuate less. What is *your* oldest sibling's middle name?

John Reisner
Air Force Institute of Technology
School of Engineering and Management
john.reisner@afit.edu

CrossTalk Now Online-only

As a reminder, CrossTalk is now completely electronic. New issues will be posted six times a year on CrossTalk's new website, <http://www.crosstalkonline.org>. Please update your browser's bookmarked CrossTalk URL to reflect the new web address. If you are currently subscribing to CrossTalk's RSS Feed, please note the feed URL has also changed to <http://www.crosstalkonline.org/issues/rss.xml>.

Each new issue will be available online both as a downloadable PDF file and also as a Flash-based digital flipbook viewable within a browser and designed to mimic the look and feel of a printed magazine.

This change reduces our carbon footprint and allows us to bring the Journal of Defense Software to our readers in their preferred and most convenient formats. This is also CrossTalk's first step towards reaching new reader devices and enhancing the suitability of the journal for our increasing electronic readership.

To help guide the transition to other digital formats, we have posted a brief reader survey. Please take a moment to participate in the survey by clicking on the "Take the Survey" button on the <http://www.crosstalkonline.org> home page or by visiting <http://www.crosstalkonline.org/survey> directly. Data gathered from this survey will be used to help determine future CrossTalk digital and mobile formats. Your input into the future direction of CrossTalk is greatly appreciated.

Thank you for your continued support and from all of us at CrossTalk, best wishes for the New Year!

Justin T. Hill
Publisher
CrossTalk, The Journal of Defense Software Engineering